开发人员指南

vCenter Orchestrator 4.0

ZH_CN-000129-00



最新的技术文档可以从 VMware 网站下载：

http://www.vmware.com/cn/support/pubs/

VMware 网站还提供最近的产品更新信息。

您如果对本文档有任何意见或建议，请把反馈信息提交至：

docfeedback@vmware.com

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

北京办公室
北京市海淀区科学院南路 2 号
融科资讯中心 C 座南 8 层
www.vmware.com/cn

上海办公室
上海市浦东新区浦东南路 999 号
新梅联合广场 23 楼
www.vmware.com/cn

广州办公室
广州市天河北路 233 号
中信广场 7401 室
www.vmware.com/cn

目录

关于本文档

《VMware vCenter Orchestrator 开发人员指南》提供有关如何使用 VMware vCenter Orchestrator 平台开发虚拟环境的过程自动化应用程序的信息和说明。

目标读者

本文档专供使用 Orchestrator 平台开发应用程序的开发人员使用。本文档对于以下类型的开发人员尤为适用。

- 要为 Orchestrator 平台新建扩展程序的应用程序开发人员。
- 要新建构建块以自动执行某些特定过程的脚本开发人员。
- 要使用诸如简单对象访问协议 (SOAP) 和 Web 服务定义语言 (WSDL) 等技术，通过网络访问这些过程的 Web 服务应用程序开发人员。
- 要使用 Web 2.0 技术为过程创建或自定义 Web 前端的 Web 设计人员。
- 要自动执行过程以节省时间、降低风险和成本并遵守相关法规或标准实践的 IT 工作人员。

示例应用程序

本文档中介绍的示例应用程序可通过下载获取。您可以从 Orchestrator 文档主页下载这些示例的 ZIP 文件。

文档反馈

VMware 欢迎您提出宝贵建议，以便改进我们的文档。如有意见，请将反馈发送到 docfeedback@vmware.com。

技术支持和教育资源

您可以获得以下技术支持资源。有关本文档和其他文档的最新版本，请访问：
http://www.vmware.com/cn/support/pubs。

在线支持和电话支持

要通过在线支持提交技术支持请求、查看产品和合同信息以及注册您的产品，请访问 http://www.vmware.com/cn/support。

客户只要拥有相应的支持合同，就可以通过电话支持，以最快的速度获得对高优先级问题的答复。请访问
http://www.vmware.com/cn/support/phone_support.html。

支持服务项目

要了解 VMware 支持服务项目如何帮助您满足业务需求，请访问
http://www.vmware.com/cn/support/services。

VMware 专业服务

VMware 教育服务课程提供了大量实践操作环境、案例研究示例，以及用作作业参考工具的课程材料。这些课程可以通过现场指导、教室授课的方式学习，也可以通过在线直播的方式学习。关于现场试点项目及实施的最佳实践，VMware 咨询服务可提供多种服务，协助您评估、计划、构建和管理虚拟环境。要了解有关教育课程、认证计划和咨询服务的信息，请访问
http://www.vmware.com/cn/services。

VMware vCenter Orchestrator 简介 1

VMware vCenter Orchestrator 是一个开发和过程自动化平台，提供可扩展的工作流库，用于创建和运行自动化的可配置过程来管理 VMware vCenter 基础架构。

Orchestrator 公开了 vCenter Server API 中的每一步操作，以便您可以将所有这些操作整合到自动化过程中。通过 Orchestrator 的开放式插件架构，您还可以将其与其他管理解决方案集成在一起。

本章讨论了以下主题：

Orchestrator 平台的主要功能特征

Orchestrator 由三个截然不同的层组成：一个可提供耦合工具所需常用功能的耦合平台、一个集成了子系统控制功能的插件架构和一个既存过程库。Orchestrator 是一个开放式平台，它不但可以通过添加新插件和库进行扩展，还可以通过一组 API 集成到更大型的 SOAP 架构中。

下面列出了 Orchestrator 的主要功能特征。

持久性

使用生产级外部数据库存储相关信息，如过程、状况和配置信息。

集中式管理

Orchestrator 提供集中过程管理方式。通过基于应用程序服务器的平台（以及完整的版本历史记录），您可以将脚本和与过程相关的基元放在同一位置。采用这种方式，可以防止未实现版本管理和适当变更控制的脚本在您的服务器上传播。

检查点

过程的每一个步骤都将保存在数据库中，这使您可以重新启动服务器而不会丢失状态和相关环境。此功能对于长时间运行的过程特别有用。

版本管理

所有的 Orchestrator 平台对象都具有相关的版本历史记录。通过此功能，您可以在将过程分配到不同的项目阶段或位置时，执行基本的变更管理操作。

脚本引擎

Mozilla Rhino JavaScript 引擎提供一种为 Orchestrator 平台创建新的构建块的方法。该脚本引擎包含基本版本控制、变量类型检查、命名空间管理和异常情况处理等增强功能。它可在以下构建块中使用：

- 操作
- 工作流
- 策略

工作流引擎

通过工作流引擎，您可以捕获业务过程。它使用下列方法之一来逐步实现自动化：

- 库的构建块
- 客户提供的构建块
- 插件

用户、调度任务或策略均可启动工作流。

策略引擎

使用策略引擎可以监控并生成事件以应对不断变化的情况。策略可以聚合来自平台或任何插件的事件，因此，您可以处理与任何集成技术有关的不断变化的情况。

Web 2.0 前端

Web 2.0 前端可以采用多种新方式表现，具备较强的灵活性。它提供一个用户可自定义的组件库来访问 vCO 耦合对象，并且使用 Ajax 技术动态更新内容，而无需重新加载整个页面。

安全性

Orchestrator 可提供以下高级安全功能：

- 公共密钥基础架构 (PKI)，可用于签署和加密在服务器之间导入和导出的内容。
- 数字权限管理 (DRM)，可用于控制导出内容的查看、编辑和重新分配方式。
- 安全套接字层 (SSL)，可用于加密桌面客户端和服务器之间的通信和通过 HTTPS 访问 Web 前端。
- 高级访问权限管理，可用于控制对过程和由这些过程操作的对象的访问权限。

Orchestrator 用户角色和相关任务

vCenter Orchestrator 会根据以下这三种全局用户角色的特定职责提供不同的工具和界面：管理员、开发人员和最终用户。

管理员

此角色对所有的 Orchestrator 平台功能都具有完全的访问权限。基本的管理任务包括：

- 安装和配置 Orchestrator
- 管理 Orchestrator 和应用程序的访问权限
- 导入和导出软件包
- 启用和禁用 Web 视图
- 运行工作流和调度任务
- 管理导入元素的版本控制

开发人员

此角色的用户拥有访问 Orchestrator 客户端界面的权限并具有以下职责：

- 开发应用程序以扩展 Orchestrator 平台功能
- 通过自定义现有工作流和创建新的工作流实现过程自动化
- 使用 Web 2.0 技术自定义这些过程的 Web 前端

最终用户

此角色的用户只拥有访问 Web 前端的权限。他们可以运行和调度工作流和策略。

Orchestrator 架构

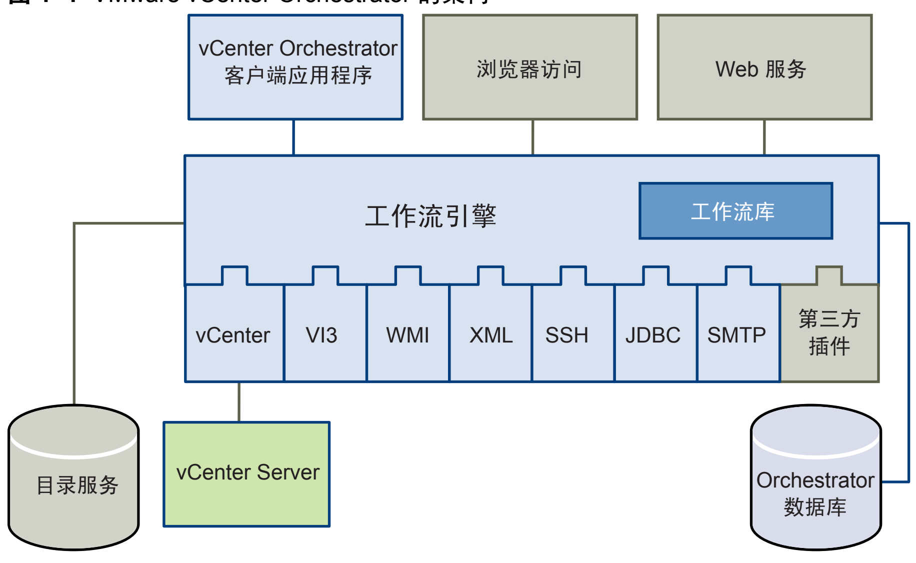
Orchestrator 包含工作流库和工作流引擎，可用于创建和运行自动执行耦合过程的工作流。可以在不同技术对象上运行工作流，Orchestrator 通过一系列插件访问这些技术。

Orchestrator 提供了一组标准插件（包括 VMware vCenter Server 4.0 的插件），使您可以耦合这些插件所在的不同环境中的任务。

此外，Orchestrator 还采用了一个开放式架构，使您可以将外部第三方应用程序插入耦合平台。您可以在由您自己定义的插件技术对象上运行工作流。Orchestrator 可以连接到目录服务服务器以管理用户帐户，此外还可以连接到数据库以存储所运行的工作流的信息。您可以通过 Orchestrator 客户端界面、Web 浏览器或 Web 服务访问 Orchestrator 及其公开的工作流和对象。

图 1–1 显示了 Orchestrator 的架构。

图 1–1 VMware vCenter Orchestrator 的架构



开发工作流 2

工作流可在 Orchestrator 客户端界面上开发。工作流开发过程需要使用工作流工作台、内置 Mozilla Rhino JavaScript 脚本引擎和 Orchestrator API。

- 工作流用户权限第 46 页，
  Orchestrator 定义了可对用户或用户组应用的权限级别。
- 运行工作流第 47 页，
  工作流按照事件的逻辑流运行。
- 开发简单示例工作流第 50 页，
  开发简单示例工作流的过程将向您展示工作流开发过程中最常见的步骤。
- 开发复杂工作流第 69 页，
  开发复杂示例工作流的过程将向您展示工作流开发过程（以及更高级的方案）中最常见的步骤，如创建自定义判定和循环。

工作流开发过程的主要阶段

工作流的开发过程涉及一系列阶段。

通常，开发工作流时会按照以下阶段顺序依次执行相关任务。

1. 提供有关工作流的一般信息。
2. 创建输入参数。
3. 通过为架构设置布局和进行链接，创建工作流的逻辑流。
4. 将每个元素的输入和输出参数与工作流属性绑定，从而在定义每个元素时生成所需的参数和属性。
5. 为可编写脚本的任务或自定义判定元素编写任何需要的脚本。
6. 通过创建工作流呈现方式，创建用户可在运行工作流时见到的输入参数对话框的布局和行为。
7. 验证工作流。

在开发期间测试工作流

您可以在开发过程中的任何时候测试工作流，即使尚未完成工作流，或未包括结束元素也可进行测试。

默认情况下，您需要在 Orchestrator 检查工作流是否有效之后才能运行该工作流。在工作流开发期间，您可以停用自动验证，以便仅运行部分工作流以进行测试。

注意 完成工作流开发后，请勿忘记重新激活自动验证功能。

步骤

1. 在 Orchestrator 客户端菜单中，单击 **Tools > User Preferences**。
2. 单击 **Workflows** 选项卡。
3. 取消选中 **Validate workflow before executing it** 复选框。

您已停用自动工作流验证。

工作流工作台

通过使用工作流工作台，可以创建和编辑工作流。工作流工作台是用于开发工作流的 Orchestrator 客户端 IDE。

您可以通过编辑现有工作流打开工作流工作台。

创建工作流

您可以在 Orchestrator 客户端界面的工作流层次结构列表中创建新的工作流。

步骤

1 在 Orchestrator 客户端中，单击 **Workflows** 视图。

2 （可选）右键单击工作流层次结构列表的根节点，或列表中的某个类别，然后选择 **Add Category** 以创建新的工作流类别。

3 （可选）提供新类别的名称。

4 右键单击该新类别或某个现有类别，然后选择 **New workflow**。

5 为新工作流命名，然后单击 **OK**。

此时将以所选类别创建新的空工作流。

下一步

您可以编辑该工作流。

在工作流工作台中编辑工作流

通过使用 Orchestrator 客户端的工作流工作台，可以编辑新的或现有的工作流。

步骤

1 在 Orchestrator 客户端中，单击 **Workflows** 视图。

2 展开工作流层次结构列表，导航到要编辑的工作流。

3 单击该工作流以进行编辑。

4 通过右键单击工作流并选择 **Edit**，打开该工作流以进行编辑。

此时工作流工作台将打开，可在其中编辑该工作流。

工作流工作台选项卡

工作流工作台由可用于编辑不同工作流组件的多个选项卡组成。

表 2–1 工作流选项板选项卡

选项卡	描述
General	可用于编辑工作流名称、提供工作流用途描述、设置版本号、定义工作流行为（如果 Orchestrator 服务器重新启动），以及定义工作流的全局属性。
Inputs	定义工作流运行时所需的参数。这些输入参数是工作流处理的数据。工作流的行为将随这些参数变化而变化。
Outputs	定义工作流运行完毕后生成的值。其他工作流或操作可在运行时使用这些值。
Schema	用于构建工作流。通过从 **Schema** 选项卡左侧的工作流选项板拖动工作流架构元素，可以构建工作流。单击架构图中的某个元素，即可在 **Schema** 选项卡的下半部分定义和编辑该元素的行为。
Presentation	用于定义当用户运行工作流时显示的用户输入对话框布局。将各种参数和属性组织整理为不同的呈现步骤和组，即可在输入参数对话框中轻松标识这些参数。您将定义用户可在呈现方式中提供的输入参数的限制。

表 2–1 工作流选项板选项卡（续）

选项卡	描述
Parameters Reference	显示哪些工作流元素会使用工作流的逻辑流属性和参数。此选项卡还将显示您在 **Presentation** 选项卡中定义的这些参数和属性的相关限制。
Executions	提供每次某个特定工作流运行时的相关详细信息。此信息包括工作流的状态、运行该工作流的用户，以及工作流开始和结束的时间和日期。
Events	提供工作流运行时发生的每个事件的相关信息。此信息包括事件的状态、运行该事件的用户，以及事件开始和结束的时间和日期。
Permissions	用于设置权限，以便与用户或用户组的工作流进行交互。

提供常规工作流信息

在工作流工作台的 **General** 选项卡上，您可以定义工作流行为的某些特定方面、设置版本号以及用户权限。

前提条件

您必须已创建工作流，并已打开该工作流的工作流工作台。

步骤

1 在工作流工作台中单击 **General** 选项卡。

2 单击 **Version** 数字以设置工作流的版本号。

此时会打开 **Version Comment** 对话框。

3 为此版本的工作流提供注释并单击 **OK**。

例如，如果刚创建了工作流，可以添加注释**初次创建**。

4 选中 **Allowed Operations** 复选框，以定义用户可对此工作流执行的操作。

可以允许用户对工作流执行以下操作。

- 查看工作流的内容
- 将工作流添加到软件包
- 编辑工作流的元素

5 通过设置 **Server restart behavior** 值，定义在 Orchestrator 服务器重新启动时工作流的行为方式。

- 将 **Resume workflow execution** 保留默认值可使工作流从服务器停止时中断运行的位置重新启动。
- 单击 **Resume workflow execution** 并选择 **DON'T resume workflow execution (set as FAILED)**，以避免工作流在 Orchestrator 服务器重新启动时也随之重新启动。

如果工作流依赖于所运行的环境，则应当避免工作流重新启动。例如，如果工作流需要特定 vCenter Server，而您将 Orchestrator 重新配置为连接到另一个 vCenter Server，则在重新启动 Orchestrator 服务器之后重新启动工作流将会导致工作流失败。

6 在 **Description** 文本框中添加对工作流的详细描述。

7 单击工作流工作台底部的 **Save**。

显示在工作流工作台左下方的绿色消息可用于确定更改已保存。

您已定义了工作流行为的某些方面，设置了版本，并且定义了用户可对工作流执行的操作。

下一步

必须定义工作流属性和参数。

定义属性和参数

创建工作流之后，必须确定工作流的全局属性以及输入和输出参数。

工作流属性是在工作流内部处理的数据。工作流输入参数是来自外部源的数据，如来自用户或另一个工作流。工作流输出参数是工作流结束时所传输的数据。

- 定义工作流属性第 15 页，

  工作流属性就是工作流要处理的数据。

- 定义工作流参数第 16 页，

  输入和输出参数可用于将信息和数据传入和传出工作流。

定义工作流属性

工作流属性就是工作流要处理的数据。

注意 创建工作流架构时，还可以在工作流架构元素中定义工作流属性。当您创建用于处理属性的工作流架构元素时，通常可以更轻松地定义属性。

前提条件

您必须已创建工作流，并已打开该工作流的工作流工作台。

步骤

1 在工作流工作台中单击 **General** 选项卡。

  属性窗格会显示在 **General** 选项卡的下半部。

2 右键单击属性窗格，然后选择 **Add Attribute**。

  新属性会显示在属性列表中，以“String”作为其默认类型。

3 单击属性名称以对名称进行更改。

  默认名称为 `attr`<X>，其中 <X> 是数字。

  注意 工作流属性的名称不能与任何工作流参数的名称相同。

4 单击类型 **Not set** 链接以从建议的列表中选择一种属性类型。

5 单击属性的类型值，将其从“String”更改为可能值列表中的其他值。

6 在 **Description** 文本框中添加属性的描述。

7 如果属性是常量而不是变量，则单击此属性名称左侧的复选框以将它的值设置为只读。

  锁定图标 ( ) 用于标识只读复选框所在的列。

8 （可选） 如果决定属性应当是输入或输出参数而不是属性，请右键单击此属性，并选择 **Move as INPUT/OUTPUT parameter** 以将属性更改为参数。

您已为工作流定义了一个属性。

下一步

可以定义工作流的输入和输出参数。

定义工作流参数

输入和输出参数可用于将信息和数据传入和传出工作流。

您可以在工作流工作台中定义工作流的参数。输入参数是用户在运行工作流时提供的数据，工作流将基于该数据执行操作。输出参数是工作流的运行结果。

前提条件

您必须已创建工作流，并已打开该工作流的工作流工作台。

步骤

1 单击工作流工作台中的相应选项卡。

   - 单击 **Inputs** 创建输入参数。
   - 单击 **Outputs** 创建输出参数。

2 右键单击参数选项卡，然后选择 **Add Parameter**。

   新参数会显示在属性列表中，以"String"作为其默认类型。

3 单击参数名称以对名称进行更改。

   输入参数的默认名称为 `arg_in`<X>，而输出参数的默认名称为 `arg_out`<X>，其中 <X> 是数字。

4 单击参数类型值，将其从"String"更改为可能值列表中的其他值。

5 在 **Description** 文本框中添加参数的描述。

6 （可选）如果稍后决定参数应当是属性而不是参数，请右键单击此参数，然后选择 **Move as attribute** 以将参数更改为属性。

您已定义工作流的输入或输出参数。

下一步

在定义工作流的参数之后，构建工作流架构。

工作流架构

工作流架构是工作流的图形表示，它采用相互连接的工作流元素流程图形式来显示工作流。

- 查看工作流架构第 17 页，

  在 Orchestrator 客户端中，您可以在某个工作流的 **Schema** 选项卡中查看该工作流的工作流架构。

- 在工作流架构中构建工作流第 17 页，

  工作流架构由一系列架构元素组成。工作流架构元素是工作流的构建块，可以代表判定、脚本式任务、操作、异常处理程序甚至其他工作流。

- 架构元素第 18 页，

  工作流工作台在 **Schema** 选项卡中通过菜单展现工作流架构元素。

- 架构元素属性第 21 页，

  架构元素中的某些属性可在工作流选项板的 **Schema** 选项卡中定义和编辑。

- 链接和绑定第 23 页，

  元素之间的链接可确定工作流的逻辑流。通过将输入和输出参数与工作流属性绑定，绑定可使用来自其他元素的数据填充元素。

- 判定第 27 页，

  工作流可以实施根据 `true` 或 `false` 布尔语句定义不同操作过程的判定函数。

- 异常处理第 29 页，

  异常处理可捕获架构元素运行时发生的所有错误。异常处理用于定义架构元素在错误发生时的行为方式。

查看工作流架构

在 Orchestrator 客户端中，您可以在某个工作流的 **Schema** 选项卡中查看该工作流的工作流架构。

步骤

1 在 Orchestrator 客户端中单击 **Workflows** 视图。

2 导航到工作流层次结构列表中的某个工作流。

3 单击此工作流以在右窗格中显示有关该工作流的信息。

4 在右窗格中选择 **Schema** 选项卡。

即会看到该工作流的图形表示。

在工作流架构中构建工作流

工作流架构由一系列架构元素组成。工作流架构元素是工作流的构建块，可以代表判定、脚本式任务、操作、异常处理程序甚至其他工作流。

您可通过将架构元素从工作流工作台左侧的工作流选项板拖到工作流架构图中，在工作流工作台中构建工作流。

编辑工作流架构

通过在工作流工作台的 **Schema** 选项卡中创建架构元素序列，可以生成工作流。

前提条件

您必须已创建一个空的工作流，并已在工作流工作台中打开该工作流以进行编辑。

步骤

1 在工作流工作台中单击 **Schema** 选项卡。

2 单击 **Schema** 选项卡左侧的 **Generic** 菜单。

3 将一个架构元素从 **Generic** 菜单拖至工作流架构中。

4 双击拖到工作流架构中的元素。

   双击元素可命名该元素。必须为元素提供工作流环境中唯一的名称。

   a 在架构元素内键入适当的元素名称。

   b 按 Enter。

   不能重命名 **Waiting timer**、**Waiting event**、**End workflow** 或 **Throw exception** 元素。

5 右键单击架构中的元素，然后选择 **Copy**。

6 在架构中的适当位置单击右键，然后选择 **Paste**。

   复制和粘贴现有架构元素是将相似元素添加到架构中的一种快捷方式。所复制元素的所有设置都将出现在粘贴的元素中。您需要根据实际情况调整所粘贴元素的设置。

7 将架构元素从 **Basic**、**Log** 或 **Network** 菜单拖到工作流架构中。

   尽管来自 **Basic**、**Log** 或 **Network** 菜单的元素都是预定义的任务，您仍可以编辑它们的名称。但是，无法编辑其脚本。

8 将架构元素从 **Action & Workflow** 菜单拖到工作流架构中。

   将操作或工作流拖到工作流架构中时，会出现一个对话框，可用于搜索要插入的操作或工作流。

9 在 **Search** 文本框中输入要插入工作流中的工作流或操作的名称或部分名称。

   与搜索匹配的工作流或操作将显示在对话框中。

10 双击某个工作流或操作以将其选中。

   您已将工作流或操作插入到工作流架构中。

11 重复步骤 1 到步骤 10，直到将所有所需架构元素都添加到工作流架构中。

   工作流架构中必须至少包含一个 **End workflow** 元素，但可以拥有多个该元素。

下一步

在将所有必要元素添加到工作流架构中之后，定义这些元素的属性，并将它们链接和绑定在一起。

修改搜索结果

可以使用 **Search** 文本框查找诸如工作流或操作之类的元素。如果搜索只返回了结果中的一部分，则可以修改搜索返回的结果数。

使用搜索功能搜索元素时，显示绿色消息框表示搜索列出了所有结果，显示黄色消息框表示搜索只列出了部分结果。

步骤

1 （可选）如果您正在工作流工作台中编辑工作流，请单击 **Save and Close** 退出工作台。

2 在 Orchestrator 客户端菜单中单击 **Tools > User Preferences**。

3 单击 **General** 选项卡。

4 在 **Finder Maximum Size** 文本框中输入希望搜索返回的结果数。

5 在"User Preferences"对话框中单击 **Save and Close**。

您已修改搜索返回的结果数。

架构元素

工作流工作台在 **Schema** 选项卡中通过菜单展现工作流架构元素。

表 2-2 介绍了可用于构建工作流的所有架构元素。

表 2-2 架构元素

架构元素名称	描述	图标	图标在工作流工作台中的位置
Start Workflow	工作流的起点。所有工作流均包含此元素，并且无法从工作流架构中将其移除。工作流只能有一个开始元素。开始元素有一个输出，但没有输入。		始终显示在 **Schema** 选项卡中
Scriptable Task	您定义的通用任务。您在此元素中写入 JavaScript 函数。		**Generic** 工作流选项板
Decision	布尔函数。判定元素将获取一个输入参数，然后返回 `true` 或 `false`。元素作出的判定类型取决于输入参数的类型。判定元素允许工作流流向不同分支方向，具体取决于判定元素收到的输入参数。如果收到的输入参数与预期值一致，则工作流将继续沿某条特定路径行进。如果输入与期望值不符，则工作流会在另一条路径上继续行进。		**Generic** 工作流选项板

表 2–2 架构元素 （续）

架构元素名称	描述	图标	图标在工作流工作台中的位置
Custom Decision	布尔函数。自定义判定可以获取多个输入参数，并根据自定义脚本作出反应。将会返回 `true` 或 `false`。		**Generic** 工作流选项板
User Interaction	允许用户将新的输入参数传递到工作流中。您可以设计用户交互元素如何呈现输入参数请求，以及如何对用户可以提供的参数施加限制。可以设置权限以确定哪些用户可以提供输入参数。当正在运行的工作流行进至用户交互元素时，它将进入被动状态，并提示用户提供输入。可以设置用户应答的超时时长。工作流会根据用户传递给它的数据继续运行，或在超时时长过期后返回异常。当它等待用户响应时，工作流令牌将处于 `waiting` 状态。		**Generic** 工作流选项板
Waiting Timer	由长时间运行的工作流使用。当一个正在运行的工作流行进至 Waiting Timer 元素时，它将进入被动状态。您可以设置工作流恢复运行的绝对日期。在其等待这一日期到达期间，工作流令牌将处于 `waiting-signal` 状态。		**Generic** 工作流选项板
Waiting Event	在长时间运行的工作流中使用。当一个正在运行的工作流行进至 Waiting Event 元素时，它将进入被动状态。您可以定义工作流恢复运行之前等待的触发器事件。在其等待这一事件期间，工作流令牌将处于 `waiting-signal` 状态。		**Generic** 工作流选项板
End Workflow	工作流的终点。可以在架构中拥有多个结束元素，以表示工作流可能生成的不同结果。结束元素有一个输入，但没有输出。当工作流行进至 End Workflow 元素时，工作流令牌将进入 `completed` 状态。		**Generic** 工作流选项板
Thrown Exception	创建异常，并停止工作流。工作流架构可以显示多个此元素。异常元素有一个输入参数，此参数只能是字符串类型，并且没有输出参数。当工作流行进至 Exception 元素时，工作流令牌将进入 `failed` 状态。		**Generic** 工作流选项板
Workflow Note	可用于为工作流的各部分添加注释。可以增加注释以描绘工作流的各个部分。还可以更改注释的背景颜色以区分不同工作流区域。工作流注释仅提供可视信息以帮助您了解架构。		**Generic** 工作流选项板

表 2–2 架构元素 （续）

架构元素名称	描述	图标	图标在工作流工作台中的位置
Pre-Defined Task	一些不可编辑的脚本元素，用于执行工作流常用的标准任务。预定义了以下任务： **Basic** ■ 休眠 (Sleep) ■ 更改凭据 (Change credential) ■ 到达日期前等待 (Wait until date) ■ 等待自定义事件 (Wait for custom event) ■ 增加计数器 (Increase counter) ■ 减少计数器 (Decrease counter) ■ 增加日期的小时数 (Add hours to date) **Log** ■ 系统日志 (System log) ■ 系统警告 (System warning) ■ 系统错误 (System error) ■ 服务器日志 (Server log) ■ 服务器警告 (Server warning) ■ 服务器错误 (Server error) ■ 系统和服务器日志 (System+server log) ■ 系统和服务器警告 (System+server warning) ■ 系统和服务器错误 (System+server error) **Network** ■ HTTP 发布 (HTTP post) ■ HTTP 获取 (HTTP get) ■ 发送自定义事件 (Send custom event)		**Basic**、**Log** 和 **Network** 工作流选项板
Action	从 Orchestrator 操作库中调用操作。当工作流行进至某个操作元素时，它会调用并运行该操作。		**Action & Workflow** 工作流选项板
Workflow	同步启动另一个工作流。一旦工作流行进至其架构中的一个工作流元素，它就会将其作为自身过程的一部分运行该工作流。在调用的工作流运行完毕之前，原始工作流将停止运行。		**Action & Workflow** 工作流选项板
Asynchronous Workflows	异步启动一个工作流。当工作流行进至一个异步工作流元素时，它会启动该工作流，并继续自身的运行。原始工作流不会等待调用的工作流完成其运行，它仍将继续自身的运行。		**Action & Workflow** 工作流选项板
Schedule Workflow	创建任务以在设定时间运行工作流，然后工作流将继续其运行。		**Action & Workflow** 工作流选项板
Nested Workflows	同时启动多个工作流。可以选择嵌套本地工作流以及位于不同 Orchestrator 服务器中的远程工作流。还可以使用不同凭据运行工作流。工作流等待所有嵌套工作流完成运行之后才会继续自身的运行。		**Action & Workflow** 工作流选项板

架构元素属性

架构元素中的某些属性可在工作流选项板的 **Schema** 选项卡中定义和编辑。

编辑架构元素的全局属性

您可以在架构元素的 **Info** 选项卡中定义架构元素的全局属性。

前提条件

工作流工作台的 **Schema** 选项卡必须包含元素。

步骤

1 在工作流工作台中单击 **Schema** 选项卡。

2 通过在工作流架构中单击架构元素，选择要编辑的元素。

  此时，架构元素的属性选项卡将显示在工作流工作台的底部。

3 单击 **Info** 选项卡。

4 在 **Name** 文本框中提供架构元素的名称。

  该名称出现在工作流架构图的架构元素中。

5 单击 **Interaction** 文本框，从列表中选择描述。

  Interaction 属性允许您在说明此元素如何与工作流的外部对象进行交互的标准描述中选择。此属性仅用于提供信息。

6 （可选）单击 **Color** 更改架构元素的背景颜色。

  通过更改单个工作流元素的颜色，可以突出显示工作流的某些特定部分。

7 在 **Business Status** 文本框中提供业务状态描述。

  Business Status 属性是此元素功能的简短描述。当工作流运行时，工作流令牌会显示每个元素运行时的业务状态。此功能对于跟踪工作流状态很有用。

架构元素属性选项卡

通过单击您拖到工作流架构中的某个架构元素，可以访问该架构元素的属性。元素属性显示在工作流工作台底部的选项卡中。

不同类型的元素会显示不同的属性选项卡，如表 2‑3 所示。

表 2‑3 架构元素属性选项卡

架构元素属性选项卡	描述	适用的架构元素类型
Attributes	元素需要的来自外部源（如用户、事件或定时器）的属性。属性可以是超时限制、时间和日期、触发器或用户凭据。	■ User Interaction ■ Waiting Event ■ Waiting Timer
Decision	用于定义判定语句。判定元素收到的输入参数或与判定语句匹配，或者不匹配，最终会产生两种可能的操作过程。	Decision
End Workflow	停止工作流，可能是因为工作流已成功完成，或者是遇到错误并返回异常。	■ End ■ Exception

表 2–3 架构元素属性选项卡 （续）

架构元素属性选项卡	描述	适用的架构元素类型
Exception	此架构元素在发生异常的情况下的行为方式。	■ Action ■ Asynchronous Workflow ■ Exception ■ Nested Workflows ■ Predefined Task ■ Schedule Workflow ■ Scriptable Task ■ User Interaction ■ Waiting Event ■ Waiting Timer ■ Workflow
External Inputs	用户必须在工作流运行的特定时刻提供的输入参数。	User Interaction
IN	此元素的 IN 绑定。IN 绑定定义架构元素从该元素在工作流中的前一个元素接收输入的方式。	■ Action ■ Asynchronous Workflow ■ Custom Decision ■ Predefined Task ■ Schedule Workflow ■ Scriptable Task ■ Workflow
Info	架构元素的常规属性和描述。**Info** 选项卡显示的信息取决于架构元素的类型。	■ Action ■ Asynchronous Workflow ■ Custom Decision ■ Decision ■ Nested Workflows ■ Note ■ Predefined Task ■ Schedule Workflow ■ Scriptable Task ■ User Interaction ■ Waiting Event ■ Waiting Timer ■ Workflow
OUT	此元素的 OUT 绑定。OUT 绑定定义架构元素用于将输出参数与工作流属性或工作流输出参数绑定的方式。	■ Action ■ Asynchronous Workflow ■ Predefined Task ■ Schedule Workflow ■ Scriptable Task ■ Workflow
Presentation	定义工作流在运行时需要用户输入的情况下用户可看到的输入参数对话框的布局。	User Interaction
Scripting	显示定义此架构元素的行为的 JavaScript 函数。对于"异步工作流"、"调度工作流"和"操作"等元素，此脚本为只读。对于可编脚本任务和自定义判定元素，您可在此选项卡中编辑 JavaScript。	■ Action ■ Asynchronous Workflow ■ Custom Decision ■ Predefined Task ■ Schedule Workflow ■ Scriptable Task

表 2–3 架构元素属性选项卡 （续）

架构元素属性选项卡	描述	适用的架构元素类型
Visual Binding	以图形的方式显示如何将此架构元素的参数和属性绑定到工作流中其前和其后的元素的参数和属性。这是元素的 IN 和 OUT 绑定的另一种表示形式。	■ Action ■ Asynchronous Workflow ■ Predefined Task ■ Schedule Workflow ■ Scriptable Task ■ Workflow
Workflows	选择要嵌套的工作流。	Nested Workflows

链接和绑定

元素之间的链接可确定工作流的逻辑流。通过将输入和输出参数与工作流属性绑定，绑定可使用来自其他元素的数据填充元素。

要了解链接和绑定，必须了解工作流的逻辑流和工作流的数据流之间的差异。

工作流的逻辑流

工作流的逻辑流就是当工作流运行时从架构中的一个元素向下一个元素行进的过程。可通过链接架构中的各元素来定义工作流的逻辑流。

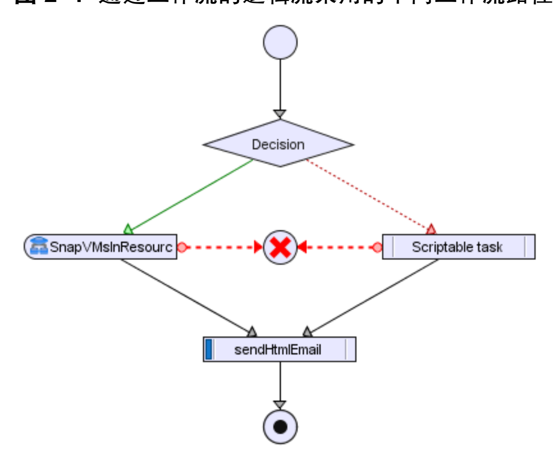
标准路径就是当所有元素都正常运行时，工作流通过逻辑流采用的行进路径。异常路径就是当某个元素未正常运行时，工作流通过逻辑流采用的行进路径。

工作流架构中具有多种不同样式的箭头，分别表示工作流可通过逻辑流采用不同路径。

- 黑色箭头表示工作流从一个元素到下一个元素采用的标准路径。
- 绿色箭头表示当布尔判定元素返回 `true` 时，工作流采用的路径。
- 由红点构成的箭头表示当布尔判定元素返回 `false` 时，工作流采用的路径。
- 由深红色点构成的箭头表示当工作流元素未正常运行时，工作流采用的异常路径。

图 2–1 显示了一个演示工作流可以采用不同路径的工作流架构的示例。

图 2–1 通过工作流的逻辑流采用的不同工作流路径



此示例工作流可以通过其逻辑流采用以下路径行进。

- 标准路径，`true` 判定结果，没有异常。

  a 判定元素返回 `true`。

  b `SnapVMsInResourcePool` 工作流成功运行。

c sendHtmlEmail 操作成功运行。

d 工作流以 completed 状态成功结束。

- 标准路径，false 判定结果，没有异常。

a 判定元素返回 false。

b 可编脚本任务元素定义的操作成功运行。

c sendHtmlEmail 操作成功运行。

d 工作流以 completed 状态成功结束。

- true 判定结果，有异常。

a 判定元素返回 true。

b SnapVMsInResourcePool 工作流遇到错误。

c 工作流返回异常，并以 failed 状态停止。

- false 判定结果，有异常。

a 判定元素返回 false。

b 可编脚本任务元素定义的操作遇到错误。

c 工作流返回异常，并以 failed 状态停止。

元素链接

链接可用于连接架构元素，同时也可用于定义工作流从一个元素流向下一个元素的逻辑流。

元素通常只能设置一个用于指向另一个工作流元素的出站链接，以及一个用于指向定义其异常行为的元素的异常链接。出站链接用于定义工作流的标准路径。异常链接则定义工作流的异常路径。大多数情况下，一个架构元素可以从多个元素接收标准路径入站链接。

对于前面的语句来说，以下元素为异常的元素。

- Start Workflow 元素收不到入站链接，并且也没有异常链接。
- Exception 元素可收到多个入站异常链接，并且没有出站或异常链接。
- Decision 元素具有两个出站链接，这些链接用于定义工作流根据判定的 true 或 false 结果采用的路径。Decision 元素没有异常链接。
- End Workflow 元素不能包含出站链接或异常链接。

创建标准路径链接

您可以使用工作流工作台的 **Schema** 选项卡中的连接器工具连接链接元素。

将一个元素链接到另一个元素时，您始终按照这些元素在工作流中的运行顺序来链接这些元素。请始终从首先运行的元素开始，在两个元素之间创建链接。

前提条件

要链接元素，必须打开工作流工作台，并且 **Schema** 必须包含元素。

步骤

1 单击 **Schema** 选项卡顶部工具栏中的连接器工具按钮，以激活连接器工具。

2 单击要链接到另一个元素的元素。

3 将指针移到突出显示的元素上方以链接到另一个元素。

元素底部即会出现一个黑色矩形。

4 在元素内靠近黑色矩形的位置单击鼠标左键，并按住鼠标左键，将指针移到目标元素上。

此时两个元素之间会出现一个箭头，并且目标元素会变成绿色。

5 释放鼠标左键。

箭头将保留在两个元素之间。

现在，已通过标准路径将这两个元素链接在一起。

下一步

元素已链接，但尚未定义数据流。必须定义 IN 和 OUT 绑定，才能将入站和出站数据与工作流属性绑定。

工作流的数据流

工作流的数据流就是当每个工作流元素运行时，其输入和输出参数与工作流属性绑定的方式。可通过使用架构元素绑定来定义工作流的数据流。

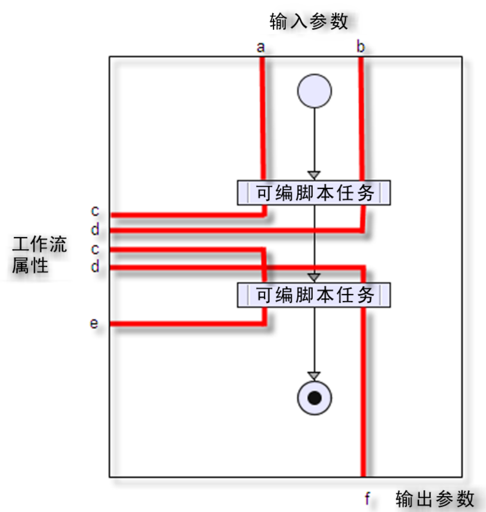
工作流架构中的元素运行时，所需要的数据形式为输入参数。它通过与创建工作流时设置的工作流属性绑定，或通过与元素运行时设置的工作流中的前一个元素的属性绑定，来获取输入参数数据。

接着，该元素会处理这些数据（必要时可能会进行转换），然后以输出参数的形式生成运行结果。该元素会将所生成的输出参数与这些新创建的工作流属性绑定。架构中的其他元素可以作为这些新工作流属性的输入参数与这些属性绑定。运行结束时，工作流可以将这些生成的属性用作输出参数。

图 2-2 显示了一个非常简单的工作流。黑色箭头表示工作流的元素链接和逻辑流。红色线条显示工作流的数据流。



图 2–2 工作流的数据流示例



数据按如下方式在工作流中行进。

1 工作流从输入参数 a 和 b 开始。

2 第一个元素处理参数 a，并将处理结果与工作流属性 c 绑定。

3 第一个元素处理参数 b，并将处理结果与工作流属性 d 绑定。

4 第二个元素获取工作流属性 c 作为输入参数并进行处理，然后将生成的输出参数与工作流属性 e 绑定。

5 第二个元素获取工作流属性 d 作为输入参数并进行处理，然后生成输出参数 f。

6 工作流结束，并生成工作流属性 f 作为输出参数，即它的运行结果。

元素绑定

必须将工作流元素的所有输入和输出参数全都与工作流属性绑定。绑定操作会在元素中设置数据，并定义元素的输出和异常行为。链接用于定义工作流的逻辑流，而绑定则用于定义数据流。

要想在元素中设置数据，在处理之后生成元素的输出参数，以及处理元素运行时可能发生的任何错误，都必须设置元素绑定。

IN 绑定	设置架构元素的入站数据。将元素的本地输入参数与源工作流属性绑定。**IN** 选项卡在 **Local Parameter** 列中列出元素的输入参数。**IN** 选项卡在 **Source Parameter** 列中列出要与其绑定本地参数的工作流属性。此选项卡还显示参数类型和参数描述。
OUT 绑定	当元素完成其运行时，更改工作流属性，并生成输出参数。**OUT** 选项卡在 **Local Parameter** 列中列出元素的输出参数。**OUT** 选项卡在 **Source Parameter** 列中列出要与其绑定本地参数的工作流属性。此选项卡还显示参数类型和参数描述。
异常绑定	如果元素在运行时遇到异常，将链接到异常处理程序。

必须使用 IN 绑定将您在架构元素中使用的每个属性或输入参数与工作流属性绑定。如果元素在运行时更改了它接收到的输入参数值，则必须通过使用 OUT 绑定将其与工作流属性绑定。通过将元素的输出参数与工作流元素绑定，可以使工作流架构中的其他后续元素将这些输出参数用作它们的输入参数。

创建工作流时常见的错误是忘记绑定输出参数值，因此无法反映元素对工作流属性所作的更改。

重要事项 如果您要添加的元素需要使用属于工作流中定义的类型的输入和输出参数，Orchestrator 会为这些参数设置绑定。必须检查 Orchestrator 绑定的参数是否正确，以防工作流定义可对其绑定元素的同类不同参数。

定义元素绑定

在链接元素以创建工作流的逻辑流之后，可定义元素绑定以定义每个元素如何处理所接收和生成的数据。

前提条件

您的工作流工作台的 **Schema** 选项卡中必须具有一个工作流架构，并且已在元素之间创建链接。

步骤

1 单击要对其设置绑定的元素。

   此时会突出显示该元素，并且在 **Schema** 选项卡的底部显示元素属性选项卡。

2 单击 **IN** 选项卡。

   IN 选项卡的内容取决于所选元素的类型。

   - 如果选择的是预定义任务、工作流或操作元素，则 **IN** 选项卡将列出该类元素的可能本地输入参数，但未设置绑定。
   - 如果选择的是另一类元素，则通过在 **IN** 选项卡中右键单击并选择 **Bind to workflow attribute/ parameter**，可以从已为该工作流定义的输入参数和属性的列表中进行选择。
   - 如果所需属性不存在，则通过在 **IN** 选项卡中右键单击并选择 **Bind to workflow attribute/parameter > Create attribute/parameter in workflow**，可以创建该属性。

3 如果存在相应参数，则选择要绑定的输入参数，并在 **Source Parameter** 文本框中单击 **Not set** 链接。

   此时将显示一个要与其绑定的可能的源参数和属性列表。

4 从建议的列表中选择一个要与本地输入参数绑定的源参数。

5 （可选）如果尚未定义要绑定的源参数，则通过在参数选取对话框中单击 **Create attribute/parameter in workflow** 链接，可以创建该参数。

6 单击 **OUT** 选项卡。

   OUT 选项卡的内容取决于所选元素的类型。

   - 如果选择的是预定义任务、工作流或操作元素，则 **OUT** 选项卡将列出该类元素的可能本地输出参数，但未设置绑定。
   - 如果选择的是另一类元素，则通过在 **OUT** 选项卡中右键单击并选择 **Bind to workflow attribute/parameter**，可以从已为该工作流定义的输出参数和属性的列表中进行选择。
   - 如果所需属性不存在，则通过在 **IN** 选项卡中右键单击并选择 **Bind to workflow attribute/parameter > Create attribute/parameter in workflow**，可以创建该属性。

7 选择要绑定的参数。

8 单击 **Source Parameter > Not set** 链接。

9 选择要与输入参数绑定的源参数。

10 （可选）如果尚未定义要绑定的参数，则通过在参数选取对话框中单击 **Create attribute/parameter in workflow** 链接，可以创建该参数。

您已定义元素接收的输入参数和它生成的输出参数，并将它们与工作流属性和参数绑定。

下一步

现在可以通过定义判定在工作流路径中创建分支。

判定

工作流可以实施根据 `true` 或 `false` 布尔语句定义不同操作过程的判定函数。

判定是工作流中的派生分支。系统会根据您、其他工作流、应用程序或在其中运行工作流的环境所提供的输入数据作出工作流判定。判定元素收到的输入参数值将决定工作流采用哪个派生分支。例如，工作流判定可能会接收给定虚拟机的电源状态作为其输入。如果该虚拟机已启动，则工作流将通过逻辑流采用某条特定路径行进。如果虚拟机已关闭，则工作流将采用另一条路径。

判定始终是布尔函数。对于每个判定来说，唯一可能的结果为 `true` 或 `false`。

自定义判定

自定义判定与标准判定的不同之处在于您将使用脚本定义该判定语句。自定义判定会根据定义的语句返回 `true` 或 `false`，如以下示例所示。

```
if (判定_语句){
    return true;
}else{
    return false;
}
```

创建判定元素链接

判定元素与其他元素的不同之处在于它们仅包含 `true` 或 `false` 输出参数。判定元素没有异常链接。

前提条件

此前必须已打开工作流工作台，并且 **Schema** 选项卡必须包含多个元素，其中至少包含一个判定元素。

步骤

1 单击判定元素以链接到另两个元素以在工作流中定义两个可能的分支。

2 单击 **Schema** 选项卡顶部工具栏中的连接器工具按钮。

3 将指针移到突出显示的判定元素上以链接到另两个元素。

- 如果将指针停留在判定元素左上方，则元素底部会显示一个绿色箭头。绿色箭头表示当判定元素收到的输入参数或属性与判定语句匹配时，工作流采用的 `true` 路径。
- 如果将指针停留在判定元素右上方，则元素底部会显示一个红色箭头。红色箭头表示当判定元素收到的输入参数或属性与判定语句不匹配时，工作流采用的 `false` 路径。

4 在判定元素的左侧单击鼠标左键，并按住左按钮，将指针移到目标元素上。

此时两个元素之间会出现一个绿色箭头，并且目标元素会变成绿色。

5 释放鼠标左键。

绿色箭头将保留在两个元素之间。您已经定义了当判定元素收到预期值时工作流采用的路径。

6 在判定元素的右侧单击鼠标左键，并按住左按钮，将指针移到目标元素上。

此时两个元素之间会出现一个红色的点状箭头，并且目标元素变成绿色。

7 释放鼠标左键。

红色点状箭头将保留在两个元素之间。您已经定义了当判定元素收到意外输入时，工作流采用的路径。

您已经定义了工作流根据判定元素收到的输入参数或属性要采取的两个可能的 `true` 或 `false` 路径。

下一步

判定元素已链接到另两个元素，但并未定义工作流如何确定采用哪个路径。必须定义判定语句。

使用判定创建工作流分支

判定元素是用于在工作流中创建分支的简单布尔函数。判定元素用于确定收到的输入数据是否与所设置的判定语句匹配。作为此判定的函数，工作流将沿着两个可能路径之一继续其行进过程。

前提条件

在定义判定之前，必须拥有链接至工作流工作台中的另两个架构元素的判定元素。

步骤

1 单击判定元素。

2 在 **Schema** 选项卡底部的元素属性选项卡中，单击 **Decision** 选项卡。

3 单击 **Not Set (NULL)** 链接为此判定选择可能的源输入参数。

此时会出现一个对话框，其中列出您在此工作流中定义的所有属性和输入参数。

4 通过双击相应的输入参数，从列表中选择一个输入参数。

5 （可选） 如果尚未定义要绑定的源参数，则通过在参数选取对话框中单击 **Create attribute/parameter in workflow** 链接，可以创建该参数。

6 从下拉菜单中选择判定语句。

 菜单建议的语句与您的系统环境相关，并且因选择的输入参数类型而异。

7 为要匹配的语句添加值。

 根据所选的输入类型和语句，可能会在值文本框中看见 **Not Set (NULL)** 链接。单击此链接可选择预定义的值。否则，例如对于字符串而言，这是用于提供值的文本框。

您已为判定元素定义了一个语句。当判定元素收到输入参数时，它会将输入参数的值与语句中的值比较，然后确定该语句的真假。

下一步

必须设置工作流处理异常的方式。

异常处理

异常处理可捕获架构元素运行时发生的所有错误。异常处理用于定义架构元素在错误发生时的行为方式。

工作流中的所有元素（判定、开始和结束元素除外）均包含一种专用于处理异常的特定输出参数类型。如果某个元素在其运行期间遇到错误，它可以向异常处理程序发送错误信号。异常处理程序会捕获该错误，并根据接收到的错误进行反应。如果所定义的异常处理程序无法处理某个特定错误，则可以将一个元素的异常输出参数与“异常”元素绑定，这样可以结束在 `failed` 状态下运行的工作流。

异常充当工作流元素中的 `try` 和 `catch` 序列。如果不需要处理元素中的给定异常，则不必绑定该元素的异常输出参数。

异常的输出参数类型始终是 `errorCode` 对象。

创建异常绑定

元素可以设置绑定，用于定义当工作流在该元素中遇到错误时的行为方式。

前提条件

工作流工作台的 **Schema** 选项卡必须包含元素。

步骤

1 单击要对其设置异常绑定的元素。

2 单击 **Schema** 选项卡顶部工具栏中的连接器工具按钮或按 Ctrl。

3 将指针移到要为其设置异常绑定的元素右侧。

 元素右侧即会出现一个红色矩形。

4 在元素内的红色矩形旁单击鼠标左键，并按住左按钮，将指针移到目标元素上。

 此时会出现一个由深红色点构成的箭头，用于链接两个元素。目标元素定义当链接到工作流的元素遇到错误时该工作流的行为。

5 单击与异常处理元素链接在一起的元素。

6 在 **Schema** 选项卡底部的架构元素属性选项卡中，单击 **Exceptions** 选项卡。

7 单击与 **Output Exception Binding** 值相关的 **Not set**。

 - 从异常属性绑定对话框中选择要与异常输出参数绑定的参数。
 - 单击 **Create parameter/attribute in workflow** 以创建异常输出参数。

8 单击用于定义异常处理行为的目标元素。

9 在 **Schema** 选项卡底部的架构元素属性选项卡中，单击 **IN** 选项卡。

10 右键单击 **IN** 选项卡，然后选择 **Bind to workflow parameter/attribute**。

11 选择异常输出参数，然后单击 **Select**。

12 在 **Schema** 选项卡底部的架构元素属性选项卡中，单击与异常处理元素相对应的 **IN** 选项卡。

13 定义异常处理元素的行为。

- 在 **OUT** 选项卡中右键单击，并选择 **Bind to workflow parameter/attribute** 为异常处理元素选择要生成的输出参数。
- 单击 **Scripting** 选项卡，并使用 JavaScript 定义异常处理元素的行为。

您定义了元素处理异常的方式。

下一步

现在必须定义当用户运行工作流时如何获取来自用户的输入参数。

在工作流启动时获取来自用户的输入参数

如果工作流需要输入参数，则它会在运行时打开一个对话框，用户需要在此输入所需的输入参数值。您可以在工作流工作台的 **Presentation** 选项卡中，组织整理此对话框的内容、布局或呈现方式。

您在 **Presentation** 选项卡中组织参数的方式会转变为工作流运行时的输入参数对话框，以及当您运行来自 Web 视图的工作流时打开的对话框。

Presentation 选项卡还可用于添加输入参数的描述，以便在用户提供输入参数时提供帮助。此外，您还可以在 **Presentation** 选项卡中设置参数的属性和限制，以限制用户所提供的参数。如果用户提供的参数不符合在 **Presentation** 选项卡中设置的限制，则工作流不会运行。

在“Presentation”选项卡中创建“Input Parameters”对话框

您将定义此对话框的布局，用户在工作流工作台的 **Presentation** 选项卡中运行工作流时会在此对话框中输入参数。

Presentation 选项卡可用于将输入参数分类，并定义这些类别在输入参数对话框中显示的顺序。

呈现方式描述

可以为在输入参数对话框中显示的每个参数或参数组添加相关描述。这些描述将向用户提供信息，可帮助他们提供正确的输入参数。您可以通过使用 HTML 格式来提供更丰富的描述文本版面。

定义呈现方式输入步骤

默认情况下，输入参数对话框会在一个列表中列出所有需要的输入参数，这些参数均按字母顺序排列。为了帮助用户输入这些输入参数，您可以在呈现方式节点中定义一些称为输入步骤的节点。这些输入步骤会将性质相似的参数分为一组。工作流运行时，输入步骤下的输入参数将在输入参数对话框的不同部分显示。

定义呈现方式显示组

每个输入步骤都可以拥有自己的节点，称为显示组。显示组用于定义参数输入文本框在其输入参数对话框部分中显示的顺序。您可以独立于输入步骤定义显示组。

创建“Input Parameters”对话框的呈现方式

您可以创建此对话框的呈现方式，用户在工作流工作台的 **Presentation** 选项卡中运行工作流时会在此对话框中输入参数。

前提条件

您必须已创建一个工作流，并且定义了一个输入参数列表。

步骤

1 在工作流工作台中，单击 **Presentation** 选项卡。

  默认情况下，工作流的所有参数均按创建顺序显示在 **Presentation** 的主节点下。

2 右键单击 **Presentation** 节点，然后选择 **New Step**。

  New Step 节点将在 **Presentation** 节点下显示。

3 双击 **New Step** 节点为其提供相应的名称，然后按 Enter。

  当工作流运行时，此名称将在输入参数对话框中显示为节头。

4 单击输入步骤，然后在 **Presentation** 选项卡下半部分的 **General** 选项卡中添加一段描述。

  此描述将显示在为用户提供信息的输入参数对话框中，以帮助他们提供正确的输入参数。您可以通过使用 HTML 格式来提供更丰富的描述文本版面。

5 右键单击您创建的输入步骤，然后选择 **Create Display group**。

  New Group 节点将在输入步骤节点下显示。

6 双击 **New Group** 节点，并为其提供相应的名称。

  当工作流运行时，此名称将在输入参数对话框中显示为子节的标头。

7 单击显示组，然后在 **Presentation** 选项卡下半部分的 **General** 选项卡中添加一段描述。

  此描述将在输入参数对话框中显示。您可以通过使用 HTML 格式来提供更丰富的描述文本版面。也可以通过使用 OGNL 语句（如 `${#param}`）将参数值添加到组描述中。

8 重复之前的步骤，直到当工作流运行时要在输入参数对话框中显示的所有输入步骤和显示组都已创建。

9 将这些参数从 **Presentation** 节点下方拖到选定的步骤和组中。

您已创建输入参数对话框的布局，用户在工作流运行时会通过此对话框提供输入参数值。

下一步

您必须设置参数属性。

设置参数属性

Orchestrator 允许您定义属性，以限定用户在运行工作流时提供的输入参数值。您定义的参数属性将对用户提供的输入参数类型和值施加限制。

每个参数均可拥有多个属性。您可以在 **Properties** 选项卡中为 **Presentation** 选项卡中的给定参数定义输入参数的属性。

参数属性将对输入参数进行验证，并修改文本框在输入参数对话框中的显示方式。某些参数属性可在参数之间建立相关性。

静态和动态参数属性值

参数属性值可为静态或动态。静态属性值保持不变。如果将某个属性值设为静态，则设置或选择来自工作流工作台根据参数类型生成的列表中的属性值。

动态属性值取决于另一个参数或属性的值。通过使用对象图形导航语言 (OGNL) 表达式，您可以定义动态属性用于获取值的函数。如果某个动态参数属性值取决于另一个参数属性值，则当另一个参数属性值更改时，OGNL 表达式将重新计算并更改该动态属性值。

设置参数属性

当工作流启动时，它会根据您设置的任意参数属性验证用户输入的参数值。

前提条件

设置参数限制之前，您必须拥有一个工作流并定义了一个输入参数列表。

步骤

1 在工作流工作台中，单击 **Presentation** 选项卡。

2 单击 **Presentation** 选项卡中的某个参数。

此参数的 **General** 和 **Properties** 选项卡将显示在 **Presentation** 选项卡的底部。

3 单击此参数的 **Properties** 选项卡。

4 右键单击 **Properties** 选项卡，然后选择 **Add property**。

此时将打开一个对话框，其中显示某个所选类型参数的可能属性列表。

5 从对话框中显示的列表中选择一个属性，然后单击 **OK**。

该属性将显示在 **Properties** 选项卡中。

6 在 **Value** 下面，通过从下拉菜单中选择相应的符号，将属性值设为静态或动态。

选项	描述
	静态属性
	动态属性

7 如果将属性值设置为静态，则根据要为其设置属性的参数类型选择属性值。

8 如果将属性值设置为动态，则通过使用 OGNL 表达式来定义函数，以获取参数属性值。

工作流工作台可在您编写 OGNL 表达式时提供帮助。

a 单击 图标可获取一个可由此表达式调用的工作流所定义的所有属性和参数组成的列表。

b 单击 图标可获取一个列出 Orchestrator API 中返回要对其定义属性的输出参数类型的所有操作的列表。

单击建议的参数和操作列表中的项目，会将这些项目添加到 OGNL 表达式中。

9 单击工作流工作台底部的 **Save**。

您已定义工作流的输入参数属性。

下一步

验证并调试工作流。

工作流输入参数属性

通过设置参数属性，可以限制用户在运行工作流时提供的输入参数。

下表列出了各类参数可能包含的属性。

表 2–4 工作流输入参数属性

参数属性	参数类型	描述
Maximum string length	字符串	设置参数的最大长度。
Minimum string length	字符串	设置参数的最小长度。
Matching regular expression	字符串	使用正则表达式验证输入。
Maximum number value	数字	设置参数的最大值。
Minimum number value	数字	设置参数的最小值。
Number format	数字	为参数输入值提供格式。
Enumeration	任意	指定按顺序排列的可能值列表。
Mandatory	任意	将该参数设置为强制性参数。
Choice from another parameter or attribute	任意	从另一个参数获得可能的用户输入值。例如，如果此参数是 `SSH:File`，而上一步中的参数是 `SSH:Folder`，则可以设置此属性以限制文件的可能输入参数值包含在 `SSH:Folder` 中。
Predefined list of elements	任意	与 **Choice from another parameter or attribute** 类似，但用户可以向由之前的参数获得的值中添加另一个值。
Show parameter input	任意	在呈现方式对话框中显示或隐藏某个参数文本框，具体取决于之前的布尔参数值。
Hide parameter input	任意	与 **Show parameter input** 类似，但取前一个布尔参数的负值。
Matching expression	从插件获取的任意参数类型	输入参数与给定表达式匹配。
Show in inventory	从插件获取的任意参数类型	设置后，只需在清单视图中右键单击任意此类型的对象，然后选择 **Execute operation**，即可在此对象上运行当前工作流。
Specify root object in selector	从插件获取的任意参数类型	如果此参数的选择器是层次结构列表选择器，则指定根对象。
Select as	从插件获取的任意参数类型	使用列表或层次结构列表选择器来选择参数。
Default value	任意	此参数的默认值。
Custom validation	OGNL 的可编脚本验证	如果调用 OGNL 表达式时返回字符串，则验证会将此字符串显示为错误结果的文本。
Auto start	布尔值	自动启动工作流。
Mandatory input	布尔值	将此参数设为强制性参数。如果不包含此属性，工作流将不会运行。

OGNL 表达式的预定义常量值

当创建 OGNL 表达式以获取动态参数属性值时，可以使用预定义常量。

Orchestrator 中定义了以下常量供 OGNL 表达式使用。

表 2–5 预定义的 OGNL 常量值

常量值	描述
`${#__current}`	自定义验证属性或与之匹配的表达式属性的当前值
`${#__username}`	启动工作流的用户的用户名

表 2–5 预定义的 OGNL 常量值 （续）

常量值	描述
`${#__userDisplayName}`	启动工作流的用户的显示名称
`${#__serverUrl}`	包含用户从中启动工作流的服务器 IP 地址的 URL
`${#__datetime}`	当前日期和时间
`${#__date}`	当前日期（时间设置为 00:00:00）
`${#__timezone}`	当前时区

工作流运行时请求用户交互

工作流运行时，有时可能需要使用来自外部来源的其他输入参数。这些输入参数可以来自另一个应用程序或工作流，或者由用户直接提供。

例如，如果在工作流运行时发生了某个特定事件，则该工作流可以请求人为交互来决定要执行的操作过程。在用户响应其信息请求或等待时间超出可能的超时时长之前，工作流将停止运行，处于等待状态。如果等待时间超过超时时长，工作流将返回异常。

用户交互有两个默认属性：`security.group` 和 `timeout.date`。通过将 `security.group` 属性设置为给定的 LDAP 用户组，即可将响应用户交互请求的权限限制为该用户组的成员。这些属性是强制性属性。

通过设置 `timeout.date` 属性，将设置该工作流等待来自用户的信息的确切截止日期和时间。如有必要，此时可以创建一个计算相对时间的工作流。

将用户交互添加到工作流

通过将用户交互架构元素添加到工作流中，可在该工作流启动之后请求用户提供输入参数。当工作流遇到用户交互架构元素时，它将挂起运行并等待用户提供所需的数据。

工作流会通过电子邮件提示用户使用输入参数对话框提供输入参数，这与工作流首次启动时获取输入参数的方法相同。您可以在用户交互架构元素属性选项卡中设置输入参数对话框的布局。

前提条件

您必须已创建一个工作流，并将其打开以便在工作流工作台中编辑，此外还必须在工作流架构中添加了一些元素。

步骤

1 将 **User Interaction** 元素拖到工作流架构中的适当位置。

2 将 **User Interaction** 元素与位于其前后的元素链接在一起。

3 单击 **User Interaction** 元素以在 **Schema** 选项卡的下半部显示其属性选项卡。

4 在 **General** 选项卡中提供用户交互的名称和描述。

5 单击 **Attributes** 以定义用户交互属性。

6 单击 `security.group` 参数的 **Not set** 链接以设置参数值。

   - 选择 **NULL** 可允许所有用户响应请求。
   - 将 `security.group` 参数设置为某个特定的 LDAP 用户组，以将响应权限限制为该组。此参数是强制性参数。

7 （可选）单击 **Create parameter/attribute in workflow** 以将 `security.group` 参数设置为某个特定的 LDAP 用户组。

   此时将显示 **Parameter information** 对话框。

   a 为参数提供一个适当的名称。

   b 选择 **Create workflow ATTRIBUTE with the same name** 以在工作流中创建 `LdapGroup` 属性。

   c 单击参数值的 **Not set** 链接。

      此时将显示 **LdapGroup** 选取框。

   d 搜索要对其限制用户交互请求的响应权限的 LDAP 用户组。

      例如，选择 **Administrators** 组意味着只有该组的成员才能响应此用户交互请求。

   e 单击 **OK**。

8 单击 `timeout.date` 参数的 **not set** 链接以设置参数值。

   - 选择 **NULL** 可使工作流无限期等待用户响应用户交互请求。
   - 设置 `timeout.date` 参数可设置工作流等待用户响应用户交互请求的截止日期和时间。

9 （可选）单击 **Create parameter/attribute in workflow** 以将 `timeout.date` 参数设置为某个特定超时日期。

   此时将显示 **Parameter information** 对话框。

   a 为参数提供一个适当的名称。

   b 选择 **Create workflow ATTRIBUTE with the same name** 以在工作流中创建 `Date` 属性。

   c 单击参数值的 **Not set** 链接。

      将显示一个日历。

   d 使用该日历可以选择工作流等待用户响应的确切截止日期和时间。

      此外，也可以将 `Date` 参数设置为计算相对日期和时间的工作流或脚本的输出。

   e 单击 **OK**。

10 单击 **External Inputs** 选项卡。

11 右键单击 **External Inputs** 选项卡，然后选择 **Bind to workflow parameter/attribute** 以定义用户必须在用户交互中提供的参数。

   - 从建议的列表中选择一个参数。
   - 如果尚未在工作流中定义此参数，请单击 **Create parameter/attribute in workflow** 以创建一个新的输入参数。

12 （可选）单击 **Exception** 选项卡。

13 （可选）以第 29 页，“创建异常绑定”中描述的方法，将异常参数定义为名为 `errorCode` 的异常字符串。

14 （可选）在元素属性选项卡中，单击 **Presentation** 选项卡。

   以第 30 页，“在“Presentation”选项卡中创建“Input Parameters”对话框”中描述的方法，定义用户将看到的输入参数对话框的布局和内容。

   注意 请注意，应在用户交互元素的 **Presentation** 选项卡中定义用户交互输入参数对话框，而不是在整个工作流的 **Presentation** 选项卡中定义。

   ---

15 单击工作流工作台底部的 **Save**。

您已定义用户交互。在此用户交互期间，工作流会等待用户提供信息，之后才会继续运行。

下一步

可以在其他工作流中调用工作流。

在工作流中调用工作流

工作流可以在其运行期间调用其他工作流。工作流可以启动另一个工作流，可能是因为它需要将该工作流的结果作为自身运行的输入参数，或者也可以启动一个工作流并让它独立地继续自己运行。工作流还可以在未来的某个给定时间启动一个工作流，或同时启动多个工作流。

- 调用工作流的工作流元素第 36 页，

  您可以使用四种方法从一个工作流中调用其他工作流。每种工作流调用方法均由一个不同的工作流架构元素表示。

- 同步调用工作流第 38 页，

  通过同步调用工作流，可使被调用的工作流作为执行调用的工作流运行的一部分运行。当执行调用的工作流运行其后续架构元素时，可以将被调用的工作流的输出参数用作输入参数。

- 异步调用工作流第 38 页，

  通过异步调用工作流，可独立于执行调用的工作流运行被调用的工作流。执行调用的工作流会继续运行，而无需等待被调用的工作流完成。

- 调度工作流第 39 页，

  可以从工作流中调用另一个工作流并将其调度为在以后的某个日期和时间启动。

- 同时调用多个工作流第 40 页，

  通过同时调用多个工作流，可使被调用的多个工作流作为执行调用的工作流运行的一部分同步运行。执行调用的工作流等待所有被调用的工作流完成之后才会继续运行。当执行调用的工作流运行其后续架构元素时，可以将被调用工作流的结果用作输入参数。

调用工作流的工作流元素

您可以使用四种方法从一个工作流中调用其他工作流。每种工作流调用方法均由一个不同的工作流架构元素表示。

同步工作流

工作流可以同步启动其他工作流。被调用的工作流将成为执行调用的工作流运行时的一个必要部分，并且与执行调用的工作流同在一个内存空间中运行。执行调用的工作流会启动另一个工作流，然后等待被调用的工作流运行结束，之后它才会开始运行其架构中的下一个元素。同步调用工作流的原因通常为，执行调用的工作流需要将被调用的工作流的输出作为后续架构元素的输入参数。例如，工作流可以调用 Start VM 工作流以启动虚拟机，然后获取此虚拟机的 IP 地址，并通过电子邮件将其传递给另一个元素或用户。

异步工作流

工作流可以异步启动其他工作流。执行调用的工作流会启动另一个工作流，但它会立即继续运行其架构中的下一个元素，而不等待被调用的工作流的结果。被调用的工作流使用执行调用的工作流定义的输入参数运行，但被调用的工作流的生命周期独立于执行调用的工作流。异步工作流可用于创建工作流链，以将输入参数从一个工作流传递到下一个工作流。例如，工作流可以在其运行期间创建各种对象。然后，该工作流可以异步启动其他工作流，将对象用作这些工作流的输入参数供其运行。当原始工作流启动了所有必需的工作流并且运行了其余的元素时，工作流结束。但是，它启动的异步工作流会独立于启动它们的工作流继续运行。

要使执行调用的工作流等待被调用工作流的结果，可以使用嵌套工作流，或者创建一个可编脚本任务，来检索被调用工作流的令牌状态。

已调度的工作流

工作流可调用其他工作流，但会将所调用工作流的启动时间延后。然后，执行调用的工作流继续运行，直到结束。通过调用已调度的工作流，可以创建一个在给定日期和时间启动该工作流的任务。当执行调用的工作流运行时，可以在 Orchestrator 客户端的 **Tasks** 和 **My Orchestrator** 视图中查看已调度的工作流。

已调度的工作流仅运行一次。通过在同步工作流中调用可编写脚本任务元素中的 `Workflow.scheduleRecurrently` 方法，您可以调度某个工作流循环运行。

嵌套工作流

工作流可通过将若干其他工作流嵌套到单一架构元素中，来同步启动这几个工作流。在执行调用的工作流行进至架构中的嵌套工作流元素时，嵌套工作流元素中列出的所有工作流将同时启动。重要的是，每个嵌套工作流都将在与执行调用的工作流不同的内存空间中启动。执行调用的工作流会等待所有嵌套工作流完成运行之后，才开始运行架构中的下一个元素。这样，它才能在运行其余元素时将嵌套工作流的结果用作输入参数。

将工作流更改传播到其他工作流

如果从一个工作流中调用另一个工作流，在您将工作流元素添加到架构中时，Orchestrator 会将该工作流的输入参数导入到执行调用的工作流中，而不是对其进行引用。

因此，如果在将被调用的工作流添加到另一个工作流之后对其进行了更改，执行调用的工作流会调用新版本的被调用工作流，而不会导入任何新的输入参数。为了避免工作流的更改对调用它们的其他工作流的行为产生影响，Orchestrator 不会自动将新的输入参数传播到执行调用的工作流中。

要将参数从一个工作流传播到其他调用它的工作流中，必须找到调用该工作流的工作流并手动同步这些工作流。

前提条件

您需要一个可供其他工作流调用的工作流。

步骤

1 修改并保存其他工作流调用的工作流。

2 关闭工作流工作台。

3 导航到在 Orchestrator 客户端的 **Workflows** 视图的层次结构列表中更改的工作流。

4 右键单击该工作流，然后选择 **References > Find Elements that Use this Element**。

   此时将显示调用此工作流的工作流列表。

5 双击列表中的一个工作流，以在 Orchestrator 客户端的 **Workflows** 视图中突出显示此工作流。

6 右键单击此工作流，然后选择 **Edit**。

   此时将打开工作流工作台。

7 在工作流工作台中单击 **Schema** 选项卡。

8 右键单击工作流架构中工作流已更改的工作流元素，然后选择 **Synchronize > Synchronize Parameters**。

9 在确认对话框中单击 **OK**。

10 保存并关闭工作流工作台。

11 对所有使用已修改工作流的工作流重复步骤 5 到步骤 10。

您已将更改后的工作流传播到调用它的其他工作流中。

同步调用工作流

通过同步调用工作流，可使被调用的工作流作为执行调用的工作流运行的一部分运行。当执行调用的工作流运行其后续架构元素时，可以将被调用的工作流的输出参数用作输入参数。

通过使用 **Workflow** 元素，可从一个工作流中同步调用另一个工作流。

前提条件

您必须已创建一个工作流，并将其打开以便在工作流工作台中编辑，此外还必须在工作流架构中添加了一些元素。

步骤

1 将 **Workflow** 元素从 **Action & Workflow** 菜单拖到工作流架构中的适当位置。

  此时将显示 **Choose workflow** 选取对话框。

2 在 **Search** 文本框中输入工作流的部分名称来搜索它。

  如果搜索只返回了结果中的一部分，请缩小搜索标准的范围或者在客户端的 **Tools > User Preferences** 菜单中增加搜索结果数量。

3 在列表中选择所需的工作流，然后单击 **OK**。

4 将 **Workflow** 元素与工作流架构中位于其前后的元素链接在一起。

5 单击 **Workflow** 元素以在 **Schema** 选项卡的下半部显示其属性选项卡。

6 在工作流架构元素的 **IN** 选项卡中将所需的输入参数与工作流绑定。

7 在工作流架构元素的 **OUT** 选项卡中将所需的输出参数与工作流绑定。

8 在 **Exceptions** 选项卡中定义工作流的异常行为。

9 单击工作流工作台底部的 **Save**。

您已从一个工作流中同步调用了另一个工作流。当工作流在其运行期间行进至同步工作流时，该同步工作流将启动，并且初始工作流要等到其完成之后才能继续运行。

下一步

可以从工作流中异步调用另一个工作流。

异步调用工作流

通过异步调用工作流，可独立于执行调用的工作流运行被调用的工作流。执行调用的工作流会继续运行，而无需等待被调用的工作流完成。

通过使用 **Asynchronous Workflow** 元素，可从一个工作流中异步调用其他工作流。

前提条件

您必须已创建一个工作流，并将其打开以便在工作流工作台中编辑，此外还必须在工作流架构中添加了一些元素。

步骤

1 将 **Asynchronous Workflow** 元素从 **Action & Workflow** 菜单拖到工作流架构中的适当位置。

  此时将显示 **Choose workflow** 选取对话框。

2 在 **Search** 文本框中输入工作流的部分名称来搜索它。

3 在列表中选择所需的工作流，然后单击 **OK**。

4 将 **Asynchronous Workflow** 元素与工作流架构中位于其前后的元素链接在一起。

5 单击 **Asynchronous Workflow** 元素以在 **Schema** 选项卡的下半部显示其属性选项卡。

6 在异步工作流元素的 **IN** 选项卡中，将所需的输入参数与工作流绑定。

7 在异步工作流元素的 **OUT** 选项卡中，绑定所需的输出参数。

   可以将输出参数与被调用的工作流绑定，或者与该工作流的结果绑定。

   - 与被调用的工作流绑定可以输出参数的形式返回该工作流。
   - 与被调用的工作流的工作流令牌绑定可返回被调用的工作流的运行结果。

8 在 **Exceptions** 选项卡中定义异步工作流元素的异常行为。

9 单击工作流工作台底部的 **Save**。

您已从一个工作流中异步调用了另一个工作流。当工作流在运行期间行进至异步工作流时，该异步工作流将启动，并且初始工作流会继续运行，而无需等待异步工作流完成。

下一步

可以将工作流调度为在以后的某个日期和时间启动。

调度工作流

可以从工作流中调用另一个工作流并将其调度为在以后的某个日期和时间启动。

通过使用 **Schedule Workflow** 元素，可在一个工作流中调度多个工作流。

前提条件

您必须已创建一个工作流，并将其打开以便在工作流工作台中编辑，此外还必须在工作流架构中添加了一些元素。

步骤

1 将 **Schedule Workflow** 元素从 **Action & Workflow** 菜单拖到工作流架构中的适当位置。

   此时将显示 **Choose Workflow** 选取对话框。

2 在 **Search** 文本框中输入工作流的部分名称来搜索它。

3 在列表中选择所需的工作流，然后单击 **OK**。

4 将 **Schedule Workflow** 元素与工作流架构中位于其前后的元素链接在一起。

5 单击 **Schedule Workflow** 元素以在 **Schema** 选项卡的下半部显示其属性选项卡。

6 单击 **IN** 属性选项卡。

   此时，要定义的属性列表中将显示一个名为 `workflowScheduleDate` 的参数，同时还显示执行调用的工作流的输入参数。

7 单击 `workflowScheduleDate` 参数的 **Not set** 链接以设置参数。

   此时将显示 **Set parameter** 对话框。

8 单击 **Create parameter/attribute in workflow** 以创建参数并设置参数的值。

   此时将打开 **Parameter information** 对话框。

9 单击 **ValueNot set** 链接以设置参数值。

   将显示一个日历。

10 使用该日历可以设置已调度工作流的启动日期和时间。

11 单击 **OK**。

12 在已调度工作流元素的 **IN** 选项卡中，将其余输入参数与已调度的工作流绑定。

13 在已调度工作流元素的 **OUT** 选项卡中，将所需的输出参数与 `Task` 对象绑定。

14 在 **Exceptions** 选项卡中定义已调度工作流元素的异常行为。

15 单击工作流工作台底部的 **Save**。

您已调度一个工作流，以使其在给定日期和时间从另一个工作流中启动。

下一步

可以在一个工作流中同时调用多个工作流。

同时调用多个工作流

通过同时调用多个工作流，可使被调用的多个工作流作为执行调用的工作流运行的一部分同步运行。执行调用的工作流等待所有被调用的工作流完成之后才会继续运行。当执行调用的工作流运行其后续架构元素时，可以将被调用工作流的结果用作输入参数。

通过使用 **Nested Workflows** 元素，可从一个工作流中同时调用多个其他工作流。可以使用嵌套工作流，来运行用户凭据与执行调用工作流不同的工作流。

前提条件

您必须已创建一个工作流，并将其打开以便在工作流工作台中编辑，此外还必须在工作流架构中添加了一些元素。

步骤

1 将 **Nested Workflows** 元素从 **Action & Workflow** 菜单拖到工作流架构中的适当位置。

  此时将显示 **Choose workflow** 选取对话框。

2 通过在 **Search** 文本框中输入其名称的一部分，搜索要启动的第一个工作流。

3 在列表中选择相应的工作流，然后单击 **OK**。

4 将 **Nested Workflows** 元素与工作流架构中位于其前后的元素链接在一起。

5 单击 **Nested Workflows** 元素以在 **Schema** 选项卡的下半部显示其属性选项卡。

6 单击 **Workflows** 架构元素属性选项卡。

  在步骤 3 中选定的工作流将出现在此选项卡中。

7 在 **Workflows** 架构元素属性选项卡的右面板的 **IN** 和 **OUT** 选项卡中，为此工作流设置 IN 和 OUT 绑定。

8 在 **Workflows** 架构元素属性选项卡的右面板中，单击 **Connection Info** 选项卡。

  通过 **Connection Info** 选项卡，您可以使用相应的凭据访问存储在非本地服务器上的工作流。

9 要访问远程服务器上的工作流，请单击 **RemoteNot set** 链接，然后提供远程服务器的主机名称或 IP 地址。

10 定义用于访问远程服务器的凭据。

  - 选择 **Inherit** 可使用与运行执行调用的工作流的用户所用凭据相同的凭据。
  - 单击 **DynamicNot set** 链接可选择 `credentials` 类型的参数在工作流的其他位置上定义的一组动态凭据。
  - 单击 **StaticNot set** 链接可直接输入凭据。

11 在 **Workflows** 选项卡中单击 **Add Workflow** 按钮，可选择更多要添加到嵌套工作流元素中的工作流。

12 重复步骤 7 到步骤 10 为所添加的每个工作流定义绑定和连接信息。

13 单击工作流架构中的嵌套工作流元素。

  元素中嵌套的工作流数将以数字的形式显示在嵌套工作流元素上。

您已从一个工作流中同时调用了多个工作流。

下一步

可以定义长时间运行的工作流。

开发长时间运行的工作流元素

处于等待状态的工作流会不断轮询要求对该工作流做出响应的对象，这会消耗系统资源。如果知道某个工作流元素在获得所需响应之前将可能等待很长时间，则可以将其作为长时间运行的元素来实施它。

每个运行中的工作流都将占用一个线程。当工作流行进至某个长时间运行的元素时，该元素会将工作流线程设置为被动状态。然后，这个长时间运行的元素会将工作流信息传递到一个单独的轮询线程，该线程用于为在服务器中运行的所有长时间运行的工作流元素轮询系统。长时间运行的工作流元素设置了一个等待期间，在此期间由长时间运行的工作流线程代表其轮询系统，而不是由每个元素自己不断尝试检索系统信息。

可以使用以下这两种方式之一设置等待期间：

- 设置一个在某个特定日期和时间之前挂起工作流的定时器，该定时器封装在 `Date` 对象中。通过在架构中包含一个 **Waiting Timer** 元素，实施基于定时器的长时间运行的工作流元素。
- 定义一个触发器事件（封装在 `Trigger` 对象中），用于在触发器事件发生之后重新启动工作流。通过在架构中添加一个 **Waiting Event** 元素或 **User Interaction** 元素，实施基于触发器的长时间运行的工作流元素。

创建 Date 对象

可以在 `Date` 对象中封装基于定时器的长时间运行的工作流需要等待的日期和时间。**Waiting Timer** 元素将此 `Date` 对象与其 `timer.date` 输入参数绑定在一起。

可以直接在 `Date` 对象中设置确切的日期和时间。当到达指定日期或指定的日期和时间满足长时间运行的工作流的条件时，工作流会重新激活并继续运行。例如，可以将工作流设置为在某个给定日期的正午重新激活。此外，您还可以创建一个根据定义的函数计算和生成 `Date` 对象的元素或工作流。

例如，以下步骤介绍了如何创建此类函数。

前提条件

您必须已创建一个工作流，并将其打开以便在工作流工作台中编辑，此外还必须在工作流架构中添加了一些元素。

步骤

1 将 **Scriptable Task** 元素从 **Generic** 菜单拖到工作流的架构中。

2 将 **Scriptable Task** 元素与工作流架构中位于其前后的元素链接在一起。

3 单击 **Scriptable Task** 元素以在 **Schema** 选项卡的下半部显示其属性选项卡。

4 在 **General** 属性选项卡中提供该函数的名称和描述。

5 在 **OUT** 属性选项卡中，创建具有下列属性的输出参数。

   a 使用值 `timerDate` 来创建 Name 属性。

   b 使用值 `Date` 来创建 Type 属性。

6 定义一个函数以在 **Scripting** 选项卡中计算并生成一个名为 `timerDate` 的 `Date` 对象。

   超时时长是以毫秒为单位的相对延迟。

   例如，通过实施以下 JavaScript 函数，可以创建 `Date` 对象。

```
timerDate = new Date();
System.log( "Current date :'" + timerDate + "'" );
timerDate.setTime( timerDate.getTime() + (50 * 1000) );
System.log( "Waiting timer will expire at '" + timerDate + "'" );
```

   以上示例 JavaScript 函数定义了一个 `Date` 对象，使用 `getTime` 方法获取当前日期和时间，并在此基础上再增加 50 秒。**Scriptable Task** 元素会生成此值作为输出参数，您可以将其与 **Waiting Timer** 元素绑定在一起。当工作流行进至 **Waiting Timer** 元素时，它将挂起并等待 50 秒后再继续运行。

7 单击工作流工作台底部的 **Save**。

您已创建了一个计算并生成 `Date` 对象的函数。**Waiting Timer** 元素可以接收此 `Date` 对象作为输入参数，以便在到达此对象中封装的日期之前挂起长时间运行的工作流。

下一步

必须将 **Waiting Timer** 元素添加到工作流中，才能实施基于定时器的长时间运行的工作流。

创建基于定时器的长时间运行的工作流

如果知道某个工作流将必须等待来自外部来源的响应，并且可以预测此等待时间，则可以将其作为基于定时器的长时间运行的工作流来实施。基于定时器的长时间运行的工作流必须等到给定日期和时间之后才会恢复运行。

通过使用 **Waiting Timer** 元素，可将工作流作为基于定时器的长时间运行的工作流来实施。

前提条件

您必须已创建一个工作流，并将其打开以便在工作流工作台中编辑，此外还必须在工作流架构中添加了一些元素。

步骤

1 将 **Waiting Timer** 元素从 **Generic** 菜单拖到工作流架构中工作流挂起运行的位置。

2 将 **Waiting Timer** 元素与工作流架构中位于其前后的元素链接在一起。

   如果实施的是用于计算日期和时间的可编写脚本的任务，则此元素必须是紧挨着 **Waiting Timer** 元素的前一个元素。

3 单击 **Waiting Timer** 元素以在 **Schema** 选项卡的下半部显示其属性选项卡。

4 在 **General** 属性选项卡中提供实施定时器的原因描述。

5 单击 **Attributes** 属性选项卡。

   此时 `timer.date` 参数将显示在属性列表中。

6 单击 `timer.date` 参数的 **Not set** 链接，以将此参数与相应的 `Date` 对象绑定。

   此时将打开 **Waiting Timer** 选取对话框，其中显示一个可能的绑定列表。

   - 从建议的列表中选择预定义的 `Date` 对象，例如一个由工作流中其他位置的 **Scriptable Task** 元素定义的对象。
   - 或者，您也可以创建一个用于设置工作流等待的具体日期和时间的 `Date` 对象。

7 （可选）创建用于设置工作流等待的具体日期和时间的 `Date` 对象。

   a 在 **Waiting Timer** 选取对话框中单击 **Create parameter/attribute in workflow**。

      此时将显示 **Parameter information** 对话框。

   b 为参数提供一个适当的名称。

   c 将类型设置保留为 `Date`。

   d 单击 **Create workflow ATTRIBUTE with the same name**。

   e 单击 **Value** 属性的 **Not set** 链接以设置参数值。

      将显示一个日历。

   f 使用该日历可以设置工作流重新启动的日期和时间。

   g 单击 **OK**。

8 单击工作流工作台底部的 **Save**。

您已为基于定时器的长时间运行工作流定义了一个定时器，该工作流在设定的日期和时间之前将挂起。

下一步

可以创建一个等待触发器事件才能继续运行的长时间运行的工作流。

创建 Trigger 对象

Trigger 对象用于监控插件定义的事件触发器。例如，vCenter Server 插件将这些事件定义为 `Task` 对象。任务结束时，触发器将向等待中的基于触发器的长时间运行工作流元素发送一条消息，以重新启动工作流。

基于触发器的长时间运行工作流等待的耗时事件必须返回一个 `VC:Task` 对象。例如，启动虚拟机的 `startVM` 操作将返回一个 `VC:Task` 对象，以便工作流中的后续元素可以监控其进度。基于触发器的长时间运行工作流的触发器事件需要此 `VC:Task` 对象作为输入参数。

在 **Scriptable Task** 元素的 JavaScript 函数中创建一个 `Trigger` 对象。此 **Scriptable Task** 元素可以是等待触发器事件的基于触发器的长时间运行工作流的一部分，或者，它也可以是向基于触发器的长时间运行工作流提供输入参数的其他工作流的一部分。触发器函数必须从 Orchestrator API 实施 `createEndOfTaskTrigger()` 方法。

重要事项 必须定义所有触发器的超时时长，否则工作流会无限期等待。

前提条件

您必须已创建一个工作流，并将其打开以便在工作流工作台中编辑，此外还必须在工作流架构中添加了一些元素。工作流必须将一个 `VC:Task` 对象声明为属性或输入参数，如来自启动或克隆虚拟机的工作流或工作流元素的 `VC:Task` 对象。

步骤

1 将 **Scriptable Task** 元素从 **Generic** 菜单拖到工作流的架构中。

2 将 **Scriptable Task** 元素与工作流架构中位于其前后的元素链接在一起。

   位于 **Scriptable Task** 前面的其中一个元素必须生成一个 `VC:Task` 对象作为输出参数。

3 单击 **Scriptable Task** 元素以在 **Schema** 选项卡的下半部显示其属性选项卡。

4 在 **Info** 属性选项卡中提供触发器的名称和描述。

5 单击 **IN** 属性选项卡。

6 右键单击 **IN** 选项卡，然后选择 **Bind to workflow parameter/attribute**。

   此时将打开输入参数选取对话框。

7 选择或创建一个类型为 **VC:Task** 的输入参数。

   此 `VC:Task` 对象表示另一个工作流或元素启动的耗时事件。

8 （可选）选择或创建 Number 类型的输入参数以定义超时时长（以秒为单位）。

9 单击 **OUT** 属性选项卡。

10 右键单击 **OUT** 选项卡，然后选择 **Bind to workflow parameter/attribute**。

   此时将打开输出参数选取对话框。

11 创建具有以下属性的输出参数。

   a 使用值 `trigger` 来创建 Name 属性。

   b 使用值 `trigger` 来创建 Type 属性。

   c 单击 **Create ATTRIBUTE with same name** 以创建该属性。

   d 将该值保留为 **Not set**。

12 在 **Exceptions** 属性选项卡中定义任何异常行为。

13 定义一个函数以在 **Scripting** 选项卡中生成 `Trigger` 对象。

   例如，通过实施以下 JavaScript 函数，可以创建 `Trigger` 对象。

```
trigger = task.createEndOfTaskTrigger(timeout);
```

   `createEndOfTaskTrigger()` 方法将返回一个监控名为 `task` 的 `VC:Task` 对象的 `Trigger` 对象。

14 单击工作流工作台底部的 **Save**。

您已定义了一个为基于触发器的长时间运行工作流创建触发器事件的工作流元素。此触发器元素生成了一个 `Trigger` 对象作为输出参数，您可以将其与 **Waiting Event** 元素绑定在一起。

下一步

必须将此触发器事件与基于触发器的长时间运行工作流中的 **Waiting Event** 元素绑定。

创建基于触发器的长时间运行的工作流

如果知道某个工作流将必须在其运行期间等待来自外部来源的响应，但不知道需要等待多长时间，则可以将其作为基于触发器的长时间运行的工作流来实施。基于触发器的长时间运行的工作流将等待定义的触发器事件发生之后才会恢复运行。

通过使用 **Waiting Event** 元素，可将工作流作为基于触发器的长时间运行的工作流来实施。当基于触发器的长时间运行的工作流行进至 **Waiting Event** 元素时，它将挂起运行，并在收到来自触发器的消息之前处于被动等待状态。等待期间，被动工作流不会占用线程，而是将工作流信息传递到一个单独的线程，该线程用于监控服务器中所有长时间运行的工作流。

前提条件

您必须已创建一个工作流，并将其打开以便在工作流工作台中编辑，此外还必须在工作流架构中添加了一些元素，定义了一个触发器事件，并已将其封装在 `Trigger` 对象中。

步骤

1 将 **Waiting Event** 元素从 **Generic** 菜单拖到工作流架构中希望工作流挂起运行的位置。

2 将 **Waiting Event** 元素与工作流架构中位于其前后的元素链接在一起。

   用于声明触发器的可编写脚本任务必须是紧挨着 **Waiting Event** 元素的前一个元素。

3 单击 **Waiting Event** 元素以在 **Schema** 选项卡的下半部显示其属性选项卡。

4 在 **General** 属性选项卡中提供等待原因的描述。

5 单击 **Attributes** 属性选项卡。

此时 `trigger.ref` 参数将显示在属性列表中。

6 单击 `trigger.ref` 参数的 **Not set** 链接，以将此参数与相应的 `Trigger` 对象绑定。

此时将打开 **Waiting Event** 选取对话框，其中显示一个可能的绑定参数列表。

7 从建议的列表中选择一个预定义的 `Trigger` 对象。

此 `Trigger` 对象表示另一个工作流或工作流元素定义的触发器事件。

8 在 **Exceptions** 属性选项卡中定义任何异常行为。

9 单击工作流工作台底部的 **Save**。

您已定义了一个挂起基于触发器的长时间运行工作流的工作流元素，该元素将等待特定触发器事件发生之后才会重新启动。

下一步

可以运行一个工作流。

配置元素

配置元素其实是一系列可在整个 Orchestrator 服务器部署中用于配置常量的属性。

在特定 Orchestrator 服务器中运行的所有工作流、操作、策略和 Web 视图，都可使用在配置元素中设置的属性。通过在配置元素中设置属性，相同的属性值可同时供 Orchestrator 服务器中运行的所有工作流、操作、策略和 Web 视图使用。

如果创建的软件包中包含使用某个配置元素属性的工作流、操作、策略或 Web 视图，则 Orchestrator 会自动在软件包中包含此配置元素，但不包含它的值。如果将包含配置元素的软件包导入另一个 Orchestrator 服务器中，此配置元素的属性值将处于未设置状态。必须使用适合于在其中导入软件包的服务器的值来设置这些属性。例如，如果创建的工作流需要使用取决于它所运行的 Orchestrator 服务器的属性值，则通过在配置元素中设置这些属性可导出该工作流，以便其他 Orchestrator 服务器可以使用它。如果直接在工作流中设置特定于服务器的属性，则将其导入另一个服务器中后该工作流可能无效，因为它可能找不到所需要的属性值。由于导入的配置元素中的属性值均未设置，因此必须使用适合于新服务器的值来设置它们。这样，配置元素才可用于在服务器之间轻松地交换工作流、操作、策略和 Web 视图。

创建配置元素

配置元素可用于设置 Orchestrator 服务器中的通用属性。服务器中运行的所有元素都可以调用在配置元素中设置的属性。通过创建配置元素，您只需在服务器中定义一次通用属性，而无需分别在每个元素中进行定义。

在 Orchestrator 客户端的 **Configurations** 视图中创建配置元素。

步骤

1 在 Orchestrator 客户端中单击 **Configurations** 视图。

2 在文件夹的层次结构列表中右键单击一个文件夹，然后选择 **New category** 创建一个新的文件夹。

3 为类别提供一个名称，然后单击 **OK**。

4 右键单击所创建的文件夹并选择 **New element**。

5 为配置元素提供一个名称，然后单击 **OK**。

6 右键单击该元素并选择 **Edit**。

此时将打开配置元素工作台。

7 通过在 **General** 选项卡中单击版本数字和提供版本注释来增大版本号。

8 在 **General** 选项卡中选中 **Allowed Operations** 复选框，以定义用户可对此配置元素执行的操作。

   可以允许用户对配置元素执行以下操作。

   - 查看配置元素的内容
   - 将配置元素添加到软件包
   - 编辑配置元素

9 在 **General** 选项卡中的 **Description** 文本框中，提供配置元素的描述。

10 单击 **Attributes** 选项卡。

11 在该选项卡中右键单击，然后选择 **Add attribute** 以创建新属性。

12 单击 **Name**、**Type**、**Value** 和 **Description** 下的属性值，以设置属性名称、类型、值和描述。

13 单击 **Permissions** 选项卡。

14 单击 **Add access rights** 为一组用户授予访问此配置元素的权限。

15 在 **Search** 文本框中搜索用户组，并从建议的列表中选择相关用户组。

16 选中相应的复选框以设置所选用户组的访问权限。

   可以对配置元素设置以下权限。

权限	描述
查看	用户可以查看配置元素，但无法查看架构或脚本。
检查	用户可以查看配置元素，包括架构和脚本。
执行	用户可以运行配置元素中的元素。
编辑	用户可以编辑配置元素中的元素。
管理	用户可以对配置元素中的元素设置权限。

17 单击 **Save and Close** 退出配置元素工作台。

您已定义了一个用于设置 Orchestrator 服务器中的通用属性的配置元素。

下一步

可以使用配置元素为工作流或操作提供属性。

工作流用户权限

Orchestrator 定义了可对用户或用户组应用的权限级别。

查看 用户可以查看工作流中的元素，但无法查看架构或脚本。

检查 用户可以查看工作流中的元素，包括架构和脚本。

执行 用户可以运行工作流。

编辑 用户可以编辑工作流。

管理 用户可以对工作流设置权限。

对工作流设置用户权限

您可以对工作流设置不同级别的权限，以便限制不同用户或用户组对该工作流的访问。

从 Orchestrator LDAP 服务器的用户和用户组中选择要为其设置权限的不同用户和用户组。

前提条件

您必须创建一个工作流，并将其打开以便在工作流工作台中编辑，此外还必须为其添加必要的元素。

步骤

1 在工作流工作台中单击 **Permissions** 选项卡。

2 单击 **Add access rights** 链接为新用户或用户组定义权限。

3 通过在 **Search** 文本框中输入文本来搜索用户或用户组。

   搜索结果会显示 Orchestrator LDAP 服务器中与搜索条件相匹配的所有用户和用户组。

4 选择用户或用户组并单击 **OK**。

5 右键单击用户并选择 **Add access rights**。

6 选中相应的复选框为此用户设置权限级别，然后单击 **OK**。

   权限级别不具叠加性。要授予用户查看工作流、检查架构和脚本、运行和编辑工作流以及更改权限的权限，您必须选中所有的复选框。

7 单击 **Save and Close** 退出软件包编辑器。

您已对工作流设置了适当的用户权限。

运行工作流

工作流按照事件的逻辑流运行。

运行工作流时，工作流中的每个架构元素均按以下顺序运行。

1 工作流将工作流令牌属性和输入参数绑定到架构元素的输入参数。

2 架构元素运行。

3 架构元素的输出参数被复制到工作流令牌属性和工作流输出参数中。

4 工作流令牌属性和输出参数存储到数据库中。

5 下一个架构元素开始运行。

对每个架构元素重复此顺序直到工作流结束。

工作流令牌检查点

工作流运行时，每个架构元素即是一个检查点。每个架构元素运行之后，Orchestrator 都会在数据库中存储工作流令牌属性，并且开始运行下一个架构元素。如果工作流意外停止，则下一次重新启动Orchestrator服务器时，将重新运行当前活动的架构元素，而且工作流将从中断发生时正在运行的架构元素的开始处继续运行。但是，Orchestrator 并不实施事务管理或回滚函数。

工作流结束

如果当前活动的架构元素是结束元素，则工作流结束。如果激活了自动验证，则没有结束元素的工作流无效，并且不会运行。当工作流行进至结束元素之后，其他工作流或应用程序即可使用该工作流的输出参数。

验证工作流

Orchestrator 提供一个工作流验证工具。验证工作流有助于识别工作流中的错误，并可检查数据是否正确地从一个元素流到下一个元素。

验证工作流时，验证工具会创建一个包含所有错误或警告的列表。单击列表中的错误可突出显示包含该错误的工作流元素。

如果在工作流工作台中运行验证工具，则该工具会对检测到的错误进行建议的快速修补。有些快速修补需要提供其他信息或输入参数，有些则会为您直接修复错误。

注意 工作流验证会检查数据是否正确地从工作流中流过，以及所有必要的链接和绑定是否到位。但它不检查由工作流中每个元素执行的数据处理操作。因此，如果架构元素中的函数不正确，则有效工作流很可能无法正确运行，并生成错误结果。

默认情况下，当您运行工作流时，Orchestrator 始终都会执行工作流验证。可以在 Orchestrator 客户端的 **Tools > User Preferences** 中更改默认验证行为。例如，有时在工作流开发期间，您可能希望运行一个已知无效的工作流以进行测试。

验证工作流和修复验证错误

可以在 Orchestrator 客户端或在工作流工作台中验证工作流。但是，如果已打开工作流以便在工作流工作台中进行编辑，则只能修复验证错误。

前提条件

要执行验证，必须拥有一个完整的工作流，该工作流具有链接的架构元素并定义了绑定。

步骤

1 单击 **Workflows** 视图。

2 导航到 **Workflows** 层次结构列表中的某个工作流。

3 （可选）右键单击此工作流，然后选择 **Validate workflow**。

   如果工作流有效，将显示一条确认消息。如果工作流无效，则会显示一个错误列表。

4 （可选）关闭 **Workflow Validation** 对话框。

5 右键单击工作流并选择 **Edit** 以打开工作流工作台。

6 单击 **Schema** 选项卡。

7 在 **Schema** 选项卡工具栏中单击 **Validate** 按钮。

   如果工作流有效，将显示一条确认消息。如果工作流无效，则会显示一个错误列表。

8 如果该工作流无效，请单击错误消息。

   验证工具将通过为其添加一个红色图标来突出显示出现错误的架构元素。如果可能，验证工具会建议执行一次 **Quick fix action**。

   - 如果同意执行建议的 **Quick fix action**，则单击它即可执行该操作。
   - 如果不同意执行建议的 **Quick fix action**，请关闭 **Workflow Validation** 对话框，然后手动修复架构元素。

   始终选中 Orchestrator 建议的“Quick Fix”是恰当之选。例如，当属性确实未正确绑定时，建议的操作可能是删除未使用的属性。

9 重复上述步骤，直到所有验证错误已消除。

您已验证了工作流，并且可能已修复了所有验证错误。

下一步

可以运行该工作流。

运行工作流

在创建并验证工作流之后，可以运行它。

此过程通过使用 Orchestrator 库中的现有工作流 **Create VM (Simple)** 来介绍如何运行工作流。

前提条件

必须拥有一个有效的工作流。

步骤

1. 在 Orchestrator 客户端中单击 **Workflows** 视图。
2. 在工作流层次结构列表中，打开 **Library > vCenter > Virtual Machine Management > Basic** 以导航到 **Create VM (Simple)** 工作流。
3. 右键单击 **Create VM (Simple)** 工作流并选择 **Execute Workflow**。

   此时将打开输入参数对话框。
4. 将下列信息输入到 **Execute Workflow** 输入参数对话框中，以在连接到 Orchestrator 的 vCenter Server 中创建一个虚拟机。
   a. 将虚拟机命名为 **`orchestrator-test`**。
   b. 单击 **VM Folder** 值的 **Not Set** 链接。

      将打开一个选取对话框。
   c. 不要在选取对话框的 **Search** 文本框中输入任何文本，并按 Enter。

      选取框将列出基础架构中包含的类型为 `VC:VmFolder` 的所有对象。如果搜索只返回了列表中的一部分，请缩小搜索标准的范围或者在 Orchestrator 客户端的 **Tools > User Preferences** 菜单中增加搜索结果数量。
   d. 单击选定的 `VC:VmFolder` 对象，然后单击 **Select**。
   e. 为 **Size of the new disk in GB** 和 **Memory size in MB** 输入适当的数值。
   f. 从 **Number of virtual CPUs** 下拉菜单中选择合适的 CPU 个数。
   g. 单击 **Guest OS** 值的 **Not Set** 链接，然后从建议的列表中选择一个客户机操作系统。
   h. 单击 **Host on which VM will be created** 值的 **Not Set** 链接，然后从建议的列表中选择一个主机。
   i. 单击 **Resource pool** 值的 **Not Set** 链接，然后在 vCenter Server 基础架构层次结构中导航到选定的资源池。
   j. 单击 **Network to connect to** 值的 **Not Set** 链接，然后从建议的列表中选择一个 `VC:Network` 对象。
   k. 单击 **Datastore on which the VM will be created** 值的 **Not Set** 链接，然后从建议的列表中选择一个 `VC:Datastore` 对象。
5. 单击 **Submit** 以运行工作流。

   工作流令牌将出现在 **Create VM (Simple)** 工作流下的叶节点上，显示表示工作流正在运行的图标。
6. 单击工作流令牌可查看工作流运行时的状态。
7. 在工作流令牌视图中单击 **Events** 选项卡，按照工作流令牌的过程执行操作，直到该过程完成。
8. 在 Orchestrator 客户端中单击 **Inventory** 视图。

9 在 vCenter Server 基础架构层次结构中导航到在步骤 4 中定义的资源池。

   如果列表中没有显示虚拟机，则单击刷新按钮以重新加载清单。

   `orchestrator-test` 虚拟机已出现在该资源池中。

10 （可选）在 **Inventory** 视图中右键单击 `orchestrator-test` 虚拟机，以查看可以在 `orchestrator-test` 虚拟机上运行的工作流的上下文相关列表。

11 （可选）选择 **Destroy VM** 以从清单中移除 `orchestrator-test` 虚拟机。

一个或多个工作流已成功运行。

开发简单示例工作流

开发简单示例工作流的过程将向您展示工作流开发过程中最常见的步骤。

名为 Start VM and Send Email 的示例工作流将启动 vCenter Server 中一台现有虚拟机，并向管理员发送电子邮件以确认虚拟机已启动。

示例工作流将执行以下任务：

1 提示用户输入要启动的虚拟机。

2 提示用户输入要向其通知虚拟机已启动或发生错误的人员的电子邮件地址。

3 将请求发送到 vCenter Server 以启动请求的虚拟机。

4 等待 vCenter Server 启动请求的虚拟机，并在虚拟机启动失败时返回错误。

5 等待 vCenter Server 启动虚拟机上的 VMware Tools。如果虚拟机启动失败或启动 VMware Tools 耗时过长，则返回错误。

6 验证虚拟机是否正在运行。

7 将通知电子邮件发送给相关人员，以通知他们计算机已启动，或者发生错误。

用于开发简单工作流的过程包含以下任务。

前提条件

在尝试开发此简单工作流示例之前，请阅读第 11 页，第 2 章“开发工作流”的所有其他各节。

步骤

1 创建简单工作流示例第 51 页，

   工作流开发过程的第一步就是创建工作流。

2 定义简单工作流示例参数第 52 页，

   在工作流工作台中定义工作流属性和参数。

3 创建简单工作流示例架构第 53 页，

   您可以在工作流工作台的 **Schema** 选项卡上创建工作流的架构。工作流架构包含工作流运行的元素。

4 链接简单工作流示例元素第 54 页，

   您可以在工作流工作台的 **Schema** 选项卡上链接工作流的元素。链接可定义数据在工作流中的流动方式。

5 创建工作流区域第 55 页，

   您可以通过添加不同颜色的工作流注释来突出显示工作流中的不同区域。通过创建不同的工作流区域，可使复杂的工作流架构更便于阅读和理解。

6 定义简单工作流示例判定绑定第 56 页，

   您可以在工作流工作台的 **Schema** 选项卡中将工作流的元素绑定在一起。判定绑定可定义判定元素如何比较接收到的输入参数与判定语句，并根据输入参数是否与判定语句匹配来生成输出参数。

7 绑定简单工作流示例操作元素第 56 页，

 您可以在工作流工作台的 **Schema** 选项卡中将工作流的元素绑定在一起。绑定可定义操作元素处理输入参数和生成输出参数的方式。

8 绑定简单工作流示例脚本任务元素第 59 页，

 您可以在工作流工作台的 **Schema** 选项卡中将工作流的元素绑定在一起。绑定可定义脚本任务元素处理输入参数和生成输出参数的方式。此外，您还可以将可编脚本任务元素与其 JavaScript 函数绑定。

9 定义简单示例工作流异常绑定第 66 页，

 您在工作流工作台中的 **Schema** 选项卡中定义异常绑定。异常绑定可定义元素处理错误的方式。

10 设置简单工作流示例属性的读写属性第 66 页，

 可以将参数和属性定义为只读常量或可写变量。此外，还可以对用户可为输入参数提供的值设置限制。

11 设置简单工作流示例参数属性第 67 页，

 您可以在工作流工作台的 **Presentation** 选项卡中设置参数属性。通过设置参数属性可影响参数的行为，并对参数的可能值设置限制。

12 设置简单工作流示例输入参数对话框的布局第 68 页，

 您可以在工作流工作台的 **Presentation** 选项卡中创建输入参数对话框的布局或呈现方式。输入参数对话框在用户运行工作流时打开，它是用户用来输入工作流运行所需的输入参数的工具。

13 验证和运行简单工作流示例第 69 页，

 创建工作流之后，可对其进行验证以发现任何可能的错误。如果工作流不包含错误，则可以运行它。

创建简单工作流示例

工作流开发过程的第一步就是创建工作流。

此示例将创建一个名为 Start VM and Send Email 的简单工作流。

前提条件

要创建简单工作流示例，系统中必须安装和配置有以下组件。

- vCenter 4.0，用于控制部分虚拟机，至少有一台处于关闭状态
- 访问 SMTP 服务器
- 有效的电子邮件地址

有关如何安装和配置 vCenter 的详细信息，请参见《ESX 和 vCenter Server 安装指南》。有关如何配置 Orchestrator 的详细信息，请参见《Orchestrator 4.0 安装和配置指南》。

要编写工作流，需要登录 Orchestrator，并且对服务器或要操作的工作流类别至少具有**查看**、**执行**、**检查**、**编辑**和**管理（最好具有此权限）**权限。

步骤

1 启动 Orchestrator 客户端界面。

2 使用 Orchestrator 用户名和密码登录。

3 在客户端界面的左侧单击 **Workflows**。

4 右键单击工作流层次结构列表的根，然后选择 **Add Category**。

5 将新类别命名为 `Workflow Examples`，然后单击 **OK**。

6 右键单击 **Workflow Examples** 类别，并选择 **New Workflow**。

7 将新工作流命名为 `Start VM and Send Email`，并单击 **OK**。

8 右键单击 **Start VM and Send Email** 工作流，并选择 **Edit**。

   此时将打开工作流工作台。

9 在 **General** 选项卡中，单击版本号数字以增大版本号。

   由于这是初次创建的工作流，因此可将版本设置为 **0.0.1**。

10 在 **General** 选项卡中选中 **Allowed operations** 复选框，以设置用户可在此工作流上执行的操作。

11 单击 **General** 选项卡中的 **Server restart behavior** 值，以设置工作流是否在服务器重新启动之后恢复运行。

12 在 **General** 选项卡的 **Description** 文本框中，提供工作流的用途描述。

   例如，可以添加以下描述。

   此简单工作流会启动虚拟机，并将确认电子邮件发送给 `Orchestrator` 管理员。

13 在 **General** 选项卡的底部单击 **Save**。

您已创建新工作流，但尚未定义其函数。

下一步

必须定义工作流的属性以及输入和输出参数。

定义简单工作流示例参数

在工作流工作台中定义工作流属性和参数。

前提条件

您必须已创建 Start VM and Send Email 工作流，并在工作流工作台中将其打开以便对其进行编辑。

步骤

1 在工作流工作台中单击 **Inputs** 选项卡。

2 右键单击 **Inputs** 选项卡并选择 **Add Parameter**。

   此时，“Inputs”选项卡中将显示名为 **arg_in_0** 的参数。

3 单击 **arg_in_0**。

4 在 **Choose Attribute Name** 对话框中键入名称 **`vm`** 并单击 **OK**。

5 单击 **Type** 文本框，然后在参数类型对话框中的搜索文本框中键入 **`vc:virtualm`**。

6 从建议的参数类型列表中选择 `VC:VirtualMachine`，然后单击 **Accept**。

7 在描述文本框中添加参数描述。

   例如，键入**要启动的虚拟机**。

8 重复上面的过程以创建另一个输入参数，使其具有下列值。

   - 名称：**`toAddress`**
   - 类型：字符串
   - 描述：**要通知其工作流程结果的人的电子邮件地址**

9 在 **Inputs** 选项卡的底部单击 **Save**。

您已定义工作流的输入参数。

下一步

必须创建工作流的架构。

创建简单工作流示例架构

您可以在工作流工作台的 **Schema** 选项卡上创建工作流的架构。工作流架构包含工作流运行的元素。

前提条件

您必须已创建 Start VM and Send Email 工作流，并定义了相关参数。

步骤

1 在工作流工作台中单击 **Schema** 选项卡。

2 单击 **Schema** 选项卡左侧的 **Generic** 菜单。

3 将一个判定元素拖至架构中的开始元素下方。

4 双击该判定元素，并将其名称改为 **`VM powered on?`**。

5 单击 **Action & Workflow**，并将一个操作元素拖至该判定元素下方。

此时将显示操作选取对话框。

6 在对话框的搜索文本框中键入**开始**。

7 选择 `startVM` 操作，然后单击 **Select**。

注意 如果已安装了 VMware Infrastructure 3.5 插件，则会列出两个 `startVM` 操作。选择 vCenter 4.0 版本，此版本有 `VC:Task` 的结果类型值。

---

8 重复步骤 1 到步骤 7，以将以下操作元素拖至架构中的 `startVM` 操作元素下方，一个放置在另一个之下。

`vim3WaitTaskEnd`	挂起工作流运行并以固定时间间隔轮询正在进行的 vCenter Server 任务，直到任务完成。在当前的示例中，`startVM` 操作启动虚拟机，`vim3WaitTaskEnd` 操作使工作流在虚拟机启动时等待。在虚拟机启动之后，`vim3WaitTaskEnd` 使工作流恢复运行。
`vim3WaitToolsStarted`	挂起工作流运行，并一直等待，直到 VMware Tools 已在目标虚拟机上启动。

9 单击“Generic”菜单，并拖动 `vim3WaitToolsStarted` 操作元素下方的一个可编脚本任务元素。

10 双击该可编脚本任务元素并将其重命名为 **`OK`**。

11 将另一个可编脚本任务元素拖至 `startVM` 操作元素的左侧。

将此脚本元素命名为 **`Already started`**。

12 将更多脚本元素拖至架构中，如下所示。

- 将一个脚本元素拖至 `startVM` 的右侧并命名为 **`startVM failed`**。
- 将一个脚本元素拖至 `vim3WaitTaskEnd` 的右侧并命名为 **`Timeout 1`**。
- 将一个脚本元素拖至 `vim3WaitToolsStarted` 的右侧并命名为 **`Timeout 2`**。
- 将一个脚本元素拖至 `OK` 的右侧并命名为 **`Send Email`**。
- 将一个脚本元素拖至 `Timeout 2` 的右侧并命名为 **`Send Email Failed`**。

13 将一个结束元素拖至**发送电子邮件**的右侧。

14 在工作流工作台的 **Schema** 选项卡底部单击 **Save**。

您已为工作流设置好了结构布局。

下一步

现在必须将这些工作流元素链接在一起。

链接简单工作流示例元素

您可以在工作流工作台的 **Schema** 选项卡上链接工作流的元素。链接可定义数据在工作流中的流动方式。

前提条件

您必须已创建 Start VM and Send Email 工作流，并且定义了参数并设置了架构布局。

步骤

1 在工作流工作台的 **Schema** 选项卡顶部的工具栏中，单击连接器工具按钮。

2 单击开始元素，并按住鼠标左键，将指针移到 `VM Powered On?` 判定元素上。

   您已将开始元素链接到判定元素。

3 按照下表中的说明链接其余的元素。

单击	链接到	箭头类型	描述
`VM Powered On?` 判定元素的左侧	`Already Started` 可编脚本任务元素	绿色	输入与判定语句匹配
`VM Powered On?` 判定元素的右侧	`startVM` 操作元素	红点	输入与判定语句不匹配
`startVM` 操作元素的中部	`vim3WaitTaskEnd` 操作元素	黑色	正常工作流进程
`vim3WaitTaskEnd` 操作元素的中部	`vim3WaitToolsStarted` 操作元素	黑色	正常工作流进程
`vim3WaitToolsStarted` 操作元素的中部	`OK` 可编脚本任务元素	黑色	正常工作流进程
`Already Started` 可编脚本任务元素的中部	`vim3WaitToolsStarted` 操作元素	黑色	正常工作流进程
`startVM` 操作元素的右侧	`StartVM Failed` 可编脚本任务元素	深红色点	异常处理
`vim3WaitTaskEnd` 操作元素的右侧	`Timeout 1` 可编脚本任务元素	深红色点	异常处理
`vim3WaitToolsStarted` 操作元素的右侧	`Timeout 2` 可编脚本任务元素	深红色点	异常处理
`StartVM Failed` 可编脚本任务元素的中部	`Send Email` 可编脚本任务元素	黑色	正常工作流进程
两个 `Timeout` 脚本元素的中部	`Send Email` 可编脚本任务元素	黑色	正常工作流进程
`OK` 可编脚本任务元素的中部	`Send Email` 可编脚本任务元素	黑色	正常工作流进程
`Send Email` 可编脚本任务元素的右侧	`Send Email Failed` 可编脚本任务元素	深红色点	异常处理
`Send Email` 可编脚本任务元素的中部	结束元素	黑色	正常工作流进程
`Send Email Failed` 可编脚本任务元素的中部	结束元素	黑色	正常工作流进程

4 在工作流工作台的 **Schema** 选项卡底部单击 **Save**。

您已将工作流元素链接在一起，从而定义了工作流的数据流。

下一步

您可以突出显示工作流中的不同区域。

创建工作流区域

您可以通过添加不同颜色的工作流注释来突出显示工作流中的不同区域。通过创建不同的工作流区域，可使复杂的工作流架构更便于阅读和理解。

前提条件

您必须已创建 Start VM and Send Email 工作流、设置了架构布局并已将架构元素链接在一起。

步骤

1 将工作流注释元素从 **Generic** 菜单拖至工作流工作台中。

2 将工作流注释置于 `Already started` 可编脚本任务元素的上方。

3 拖动工作流注释的边缘以调整其大小，以便其将 `Already started` 可编脚本任务元素包围在其中。

4 双击文本，并添加描述。例如，**如果虚拟机已启动，则采用此路径**。

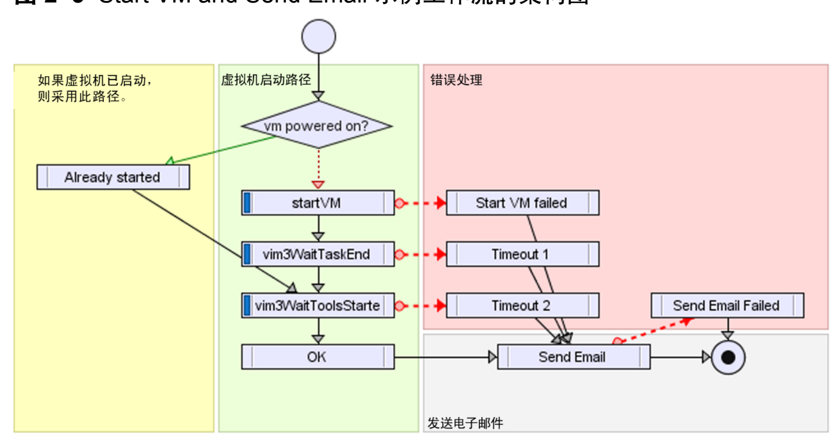
5 在工作流工作台底部的 **Info** 选项卡中单击 **Color**，然后选择背景颜色。

6 重复步骤 1 到步骤 5 以突出显示工作流中的其他区域。

   - 按照元素的垂直顺序从 `Is virtual machine on?` 判定元素到 `OK` 元素依次放置注释。添加描述**虚拟机启动路径**。
   - 将注释置于 `startVM failed`、`Timeout` 可编脚本任务元素和 `Send Email Failed` 可编脚本任务元素的上方。添加描述**错误处理**。
   - 将注释置于 `Send Email` 可编脚本任务元素的上方。

图 2-3 显示了示例工作流架构的图示。

图 2-3 Start VM and Send Email 示例工作流的架构图



下一步

定义元素参数之间的绑定。

定义简单工作流示例判定绑定

您可以在工作流工作台的 **Schema** 选项卡中将工作流的元素绑定在一起。判定绑定可定义判定元素如何比较接收到的输入参数与判定语句，并根据输入参数是否与判定语句匹配来生成输出参数。

前提条件

您必须已创建 Start VM and Send Email 工作流，并且定义了参数、设置了架构布局并已将架构元素链接在一起。

步骤

1 单击 `Is VM Powered On?` 判定元素。

2 在 **Schema** 选项卡底部的架构元素属性窗格中，单击 **Decision** 选项卡。

3 单击 **Not set (NULL)** 链接，并从建议的参数列表中选择 `vm` 作为判定元素的输入参数。

4 从下拉菜单中建议的判定语句列表中选择 **state equals** 语句。

   此时值文本框中将显示 **Not set** 链接，向您呈现有限的可能值选择。

5 选择 `poweredOn`。

6 在工作流工作台的 **Schema** 选项卡底部单击 **Save**。

您已定义了判定元素将依据其来比较所接收的输入参数值的 True 或 False 语句。

下一步

必须为工作流中的其他元素定义绑定。

绑定简单工作流示例操作元素

您可以在工作流工作台的 **Schema** 选项卡中将工作流的元素绑定在一起。绑定可定义操作元素处理输入参数和生成输出参数的方式。

前提条件

您必须已创建 Start VM and Send Email 工作流，并且定义了参数、设置了架构布局并已将架构元素链接在一起。

步骤

1 单击 `startVM` 操作元素。

2 在 **Info** 选项卡中设置以下常规信息。

Interaction	无外部交互
Color	无
Business status	选中此复选框，并添加文本**发送启动虚拟机**。
Description	保留文本**启动 / 恢复一个虚拟机。返回启动任务**

3 在 **Schema** 选项卡底部的架构元素属性窗格中，单击 **IN** 选项卡。

   此时您将看到对 `startVM` 操作可用的两个可能的输入参数：`vm` 和 `host`。

   Orchestrator 会自动将 `vm` 参数与 `vm[in-parameter]` 绑定，因为 `startVM` 操作只能将 `VC:VirtualMachine` 作为输入参数。设置工作流输入参数以便将其自动绑定到该操作时，Orchestrator 会检测到您定义的 `vm` 参数。

4 将 `host` 设置为 **NULL**。

   此参数是可选参数，因此您可以将其设置为 Null。但是，如果将其设置保留为 **Not set**，则该工作流将不会进行验证操作。

5 在架构元素属性窗格中单击 **OUT** 选项卡。

此时将显示所有操作生成的默认输出参数 actionResult。

6 单击 actionResult 参数的 **Not set** 链接。

7 单击 **Create parameter/attribute in workflow** 链接。

此时将打开 **Parameter Information** 对话框，您可以在此定义此输出参数的值。startVM 操作的输出参数类型为 VC:Task 对象。

8 将参数命名为 **powerOnTask**。

9 提供此参数的描述。

例如，**包含启动虚拟机的结果**。

10 单击 **Create workflow ATTRIBUTE with the same name**。

11 单击 **OK** 退出 **Parameter Information** 对话框。

12 重复步骤 1 到步骤 11 以将输入参数和输出参数绑定到 vim3WaitTaskEnd 和 vim3WaitToolsStarted 操作元素。

第 57 页，“简单工作流示例操作元素绑定” 列出了 vim3WaitTaskEnd 和 vim3WaitToolsStarted 操作元素的绑定。

13 在工作流工作台的 **Schema** 选项卡底部单击 **Save**。

此时，操作元素的输入和输出参数已与相应的参数类型和值绑定。

下一步

绑定可编脚本任务元素并定义其函数。

简单工作流示例操作元素绑定

绑定可定义简单工作流示例的操作元素处理输入和输出参数的方式。

定义绑定时，Orchestrator 会将已在工作流中定义的参数显示为候选的绑定对象。如果尚未在工作流中定义需要的参数，则只能选择参数 NULL。单击 **Create parameter/attribute in workflow** 创建新参数。

vim3WaitTaskEnd 操作

vim3WaitTaskEnd 操作元素可声明跟踪任务进度和轮询速率的常量。表 2-6 显示了 vim3WaitTaskEnd 操作需要的输入和输出参数绑定。

表 2–6 vim3WaitTaskEnd 操作的绑定值

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
task	IN	绑定	■ 本地参数：task ■ 源参数：task[attribute] ■ 类型：VC:Task ■ 描述：**当前正在运行的 vCenter Server 任务。**
progress	IN	创建	■ 本地参数：progress ■ 源参数：progress[attribute] ■ 类型：布尔 ■ 值：无 (false) ■ 描述：**等待 vCenter Server 任务完成时记录的进度。**
pollRate	IN	创建	■ 本地参数：pollRate ■ 源参数：pollRate[attribute] ■ 类型：数字 ■ 值：**2** ■ 描述：**vim3WaitTaskEnd 检查 vCenter Server 任务进度的轮询率（单位：秒）。**
actionResult	OUT	创建	■ 本地参数：actionResult ■ 源参数：returnedManagedObject[attribute] ■ 类型：任意 ■ 描述：**waitTaskEnd 操作返回的管理对象。**

vim3WaitToolsStarted 操作

vim3WaitToolsStarted 操作元素将等待 VMware Tools 在虚拟机上安装完成，然后定义轮询速率和超时时长。表 2-6 显示了 vim3WaitToolsStarted 操作需要的输入参数绑定。

vim3WaitToolsStarted 操作元素没有输出，因此不需要输出绑定。

表 2–7 vim3WaitToolsStarted 操作的绑定值

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
vm	IN	自动绑定	■ 本地参数：vm ■ 源参数：vm[in-parameter] ■ 类型：VC:VirtualMachine ■ 值：不可编辑，变量不是工作流属性。 ■ 描述：**要启动的虚拟机。**
pollingRate	IN	绑定	■ 本地参数：pollRate ■ 源参数：pollRate[attribute] ■ 类型：数字 ■ 描述：**vim3WaitTaskEnd 检查 vCenter Server 任务进度的轮询率（单位：秒）。**
timeout	IN	创建	■ 本地参数：timeout ■ 源参数：timeout[attribute] ■ 类型：数字 ■ 值：**10** ■ 描述：**引发异常之前，vim3WaitToolsStarted 等待的超时限制。**

绑定简单工作流示例脚本任务元素

您可以在工作流工作台的 **Schema** 选项卡中将工作流的元素绑定在一起。绑定可定义脚本任务元素处理输入参数和生成输出参数的方式。此外，您还可以将可编脚本任务元素与其 JavaScript 函数绑定。

前提条件

您必须已创建 Start VM and Send Email 工作流、定义了其参数、设置了架构布局并链接了架构元素。

步骤

1 单击 `Already Started` 可编脚本任务元素。

2 在 **Info** 选项卡中设置以下常规信息。

Interaction	无外部交互
Color	无
Business status	选中此复选框，然后添加文本**虚拟机已启动**。
Description	保留文本 **该虚拟机已启动，绕过 startVM 和 waitTaskEnd，检查虚拟机工具是否已启动并正在运行。**

3 在 **Schema** 选项卡底部的架构元素属性窗格中，单击 **IN** 选项卡。

由于这是自定义的可编脚本任务元素，因此没有预定义的属性。

4 右键单击 **IN** 选项卡，然后选择 `Bind to workflow parameter/attribute`。

5 从建议的参数列表中选择 `vm`。

6 将 **OUT** 和 **Exception** 选项卡留空。

   此元素不会生成输出参数或异常。

7 单击 **Scripting** 选项卡。

8 添加以下 JavaScript 函数。

```
//Writes the following event in the vCO database
Server.log("VM '"+ vm.name +"' already started");
```

9 重复步骤 1 到步骤 7 以将其余的输入参数与其他可编脚本任务元素绑定。

   第 60 页，“简单工作流示例可编脚本任务元素绑定” 列出了 `Start VM failed`、`Timeout or Error` 和 `OK` 可编脚本任务元素的绑定。

10 在工作流工作台的 **Schema** 选项卡底部单击 **Save**。

您已将可编脚本任务元素与其输入和输出参数绑定，并提供了定义其函数的脚本。

下一步

必须定义异常处理。

简单工作流示例可编脚本任务元素绑定

绑定可定义简单工作流示例的可编脚本任务元素处理输入参数的方式。此外，您还可以将可编脚本任务元素与其 JavaScript 函数绑定。

定义绑定时，Orchestrator 会将已在工作流中定义的参数显示为候选的绑定对象。如果尚未在工作流中定义所需的参数，则只能选择参数 `NULL`。单击 **Create parameter/attribute in workflow** 创建新参数。

Start VM Failed 可编脚本任务

通过设置有关虚拟机启动失败的电子邮件通知并将该事件写入 Orchestrator 日志，Start VM Failed 可编脚本任务元素可处理 `startVM` 操作返回的任何异常情况。

表 2-8 显示了 Start VM Failed 可编脚本任务元素需要的输入和输出参数绑定。

表 2–8 Start VM Failed 可编脚本任务元素的绑定

参数名称	绑定类型	与现有参数绑定还是创建参数？	绑定值
vm	IN	绑定	■ 本地参数：vm ■ 源参数：vm[in-parameter] ■ 类型：VC:VirtualMachine ■ 描述：**要启动的虚拟机。**
errorCode	IN	创建	■ 本地参数：errorCode ■ 源参数：errorCode[attribute] ■ 类型：字符串 ■ 描述：**捕获任何启动虚拟机的异常。**
body	OUT	创建	■ 本地参数：body ■ 源参数：body[attribute] ■ 类型：字符串 ■ 描述：**电子邮件正文**

Start VM Failed 可编脚本任务元素执行以下脚本函数。

```
body = "Unable to execute powerOnVM_Task() on VM '"+vm.name+"', exception found:"+errorCode;
//Writes the following event  in the vCO database
Server.error("Unable to execute powerOnVM_Task() on VM '"+vm.name, "Exception found:"+errorCode);
```

Timeout 1 可编脚本任务元素

通过设置有关任务失败的电子邮件通知内容并将该事件写入 Orchestrator 日志，Timeout 1 可编脚本任务元素可处理 `vim3WaitTaskEnd` 操作返回的任何异常情况。

表 2–9 显示了 Timeout 1 可编脚本任务元素需要的输入和输出参数绑定。

表 2–9 Timeout 1 可编脚本任务元素的绑定

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
vm	IN	绑定	■ 本地参数：vm ■ 源参数：vm[in-parameter] ■ 类型：VC:VirtualMachine ■ 描述：**要启动的虚拟机。**
errorCode	IN	绑定	■ 本地参数：errorCode ■ 源参数：errorCode[attribute] ■ 类型：字符串 ■ 描述：**捕获任何启动虚拟机的异常。**
body	OUT	绑定	■ 本地参数：body ■ 源参数：body[attribute] ■ 类型：字符串 ■ 描述：**电子邮件正文**

Timeout 1 可编脚本任务元素需要以下脚本函数。

```
body = "Error while waiting for poweredOnVM_Task() to complete on VM '"+vm.name+"', exception
found:"+errorCode;
//Writes the following event in the vCO database
Server.error("Error while waiting for poweredOnVM_Task() to complete on VM '"+vm.name, "Exception
found:"+errorCode);
```

Timeout 2 可编脚本任务元素

通过设置有关任务失败的电子邮件通知内容并将该事件写入 Orchestrator 日志，Timeout 2 可编脚本任务元素可处理 vim3WaitToolsStarted 操作返回的任何异常情况。

表 2‑10 显示了 Timeout 2 可编脚本任务元素需要的输入和输出参数绑定。

表 2–10 Timeout 2 可编脚本任务元素的绑定

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
vm	IN	绑定	■ 本地参数：vm ■ 源参数：vm[in-parameter] ■ 类型：VC:VirtualMachine ■ 描述：**要启动的虚拟机。**
errorCode	IN	绑定	■ 本地参数：errorCode ■ 源参数：errorCode[attribute] ■ 类型：字符串 ■ 描述：**捕获任何启动虚拟机的异常。**
body	OUT	绑定	■ 本地参数：body ■ 源参数：body[attribute] ■ 类型：字符串 ■ 描述：**电子邮件正文**

Timeout 2 可编脚本任务元素需要以下脚本函数。

```
body = "Error while waiting for VMware tools to be up on VM '"+vm.name+"', exception
found:"+errorCode;
//Writes the following event in the vCO database
Server.error("Error while waiting for VMware tools to be up on VM '"+vm.name, "Exception
found:"+errorCode);
```

OK 可编脚本任务元素

OK 可编脚本任务元素用于接收虚拟机已成功启动的通知，设置有关成功启动虚拟机的电子邮件通知内容，并将该事件写入 Orchestrator 日志中。

表 2–11 显示了 OK 可编脚本任务元素需要的输入和输出参数绑定。

表 2–11 OK 可编脚本任务元素的绑定

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
vm	IN	绑定	■ 本地参数：vm ■ 源参数：vm[in-parameter] ■ 类型：VC:VirtualMachine ■ 描述：**要启动的虚拟机。**
body	OUT	绑定	■ 本地参数：body ■ 源参数：body[attribute] ■ 类型：字符串 ■ 描述：**电子邮件正文**

OK 可编脚本任务元素需要以下脚本函数。

```
body = "The VM '"+vm.name+"' has started succesfully and is ready for use";
//Writes the following event in the vCO database
Server.log(body);
```

Send Email Failed 可编脚本任务元素

Send Email Failed 可编脚本任务元素用于接收电子邮件发送失败的通知，并将该事件写入 Orchestrator 日志中。

表 2‑11 显示了 Send Email Failed 可编脚本任务元素需要的输入和输出参数绑定。

表 2‑12 Send Email Failed 可编脚本任务元素的绑定

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
vm	IN	绑定	■ 本地参数：vm ■ 源参数：vm[in-parameter] ■ 类型：VC:VirtualMachine ■ 描述：**要启动的虚拟机。**
toAddress	IN	绑定	■ 本地参数：toAddress ■ 源参数：toAddress[in-parameter] ■ 类型：字符串 ■ 描述：**要通知其工作流程结果的人的电子邮件地址**
emailErrorCode	IN	创建	■ 本地参数：emailErrorCode ■ 源参数：emailErrorCode[attrbute] ■ 类型：字符串 ■ 描述：**捕获任何发送电子邮件的异常**

Send Email Failed 可编脚本任务元素需要以下脚本函数。

```
//Writes  the following event in the vCO database
Server.error("Couldn't send result email to '"+toAddress+"' for VM '"+vm.name, "Exception 
found:"+emailErrorCode);
```

Send Email 可编脚本任务元素

Start VM and Send Email 工作流的目的是在其启动虚拟机时通知管理员。为此，必须定义用于发送电子邮件的可编脚本任务。为了发送电子邮件，Send Email 可编脚本任务元素需要具有 SMTP 服务器、电子邮件发件人和收件人的地址、电子邮件主题和电子邮件内容。

表 2‑11 显示了 Send Email 可编脚本任务元素需要的输入和输出参数绑定。

表 2–13 Send Email 可编脚本任务元素的绑定

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
vm	IN	绑定	■ 本地参数：vm ■ 源参数：vm[in-parameter] ■ 类型：VC:VirtualMachine ■ 描述：**要启动的虚拟机。**
toAddress	IN	绑定	■ 本地参数：toAddress ■ 源参数：toAddress[in-parameter] ■ 类型：字符串 ■ 描述：**要通知其工作流程结果的人的电子邮件地址**
body	IN	绑定	■ 本地参数：body ■ 源参数：body[attribute] ■ 类型：字符串 ■ 描述：**电子邮件正文**
smtpHost	IN	创建	■ 本地参数：smtpHost ■ 源参数：smtpHost[attribute] ■ 类型：字符串 ■ 描述：**电子邮件 SMTP 服务器**
fromAddress	IN	创建	■ 本地参数：fromAddress ■ 源参数：fromAddress[attribute] ■ 类型：字符串 ■ 描述：**发件人的电子邮件地址**
subject	IN	创建	■ 本地参数：subject ■ 源参数：subject[attribute] ■ 类型：字符串 ■ 描述：**电子邮件主题**

Send Email 可编脚本任务元素需要以下脚本函数。

```
//Create an instance of EmailMessage
var myEmailMessage = new EmailMessage() ;

//Apply methods on this instance that populate the email message
myEmailMessage.smtpHost = smtpHost;
myEmailMessage.fromAddress = fromAddress;
myEmailMessage.toAddress = toAddress;
myEmailMessage.subject = subject;
myEmailMessage.addMimePart(body , "text/html");
```

```
//Apply the method that sends the email message
myEmailMessage.sendMessage();
System.log("Sent email to '"+toAddress+"'");
```

定义简单示例工作流异常绑定

您在工作流工作台中的 **Schema** 选项卡中定义异常绑定。异常绑定可定义元素处理错误的方式。

工作流中的以下元素返回了异常：`startVM`、`vim3WaitTaskEnd` 和 `vim3WaitToolsStarted`。

前提条件

您必须已创建 Start VM and Send Email 工作流，并且定义了参数并设置了架构布局。

步骤

1 单击 `startVM` 操作元素。

2 单击 **Schema** 选项卡底部的 **Exceptions** 选项卡。

3 单击 **Not set** 链接。

4 从建议的列表中选择 `errorCode`。

5 重复步骤 1 到步骤 4，为 `vim3WaitTaskEnd` 和 `vim3WaitToolsStarted` 设置到 `errorCode` 的异常绑定。

6 单击 `Send Email` 可编脚本任务元素。

7 单击 **Schema** 选项卡底部的 **Exceptions** 选项卡。

8 单击 **Not set** 链接。

9 从建议的列表中选择 `emailErrorCode`。

10 在工作流工作台的 **Schema** 选项卡底部单击 **Save**。

您已为返回异常的元素定义了异常绑定。

下一步

必须对属性和参数设置读写属性。

设置简单工作流示例属性的读写属性

可以将参数和属性定义为只读常量或可写变量。此外，还可以对用户可为输入参数提供的值设置限制。

通过将某些参数设置为只读，可以使其他开发人员不中断工作流核心功能即可改写或修改工作流。

前提条件

您必须已创建工作流、为架构设置了布局并进行了链接并为所有元素定义了 **IN**、**OUT** 和异常绑定。

步骤

1 在工作流工作台的顶部单击 **General** 选项卡。

   在 **Attributes** 的下方是您定义的所有属性的列表，每个属性旁都有一个复选框。选中这些复选框即会将相应属性设置为只读。

2 选中复选框以将下列属性设置为只读常量：

   - `progress`
   - `pollRate`
   - `timeout`
   - `smtpHost`
   - `fromAddress`
   - `subject`

您已定义哪些工作流属性为常量，哪些为变量。

下一步

必须设置参数属性，并对该参数的可能值设置限制。

设置简单工作流示例参数属性

您可以在工作流工作台的 **Presentation** 选项卡中设置参数属性。通过设置参数属性可影响参数的行为，并对参数的可能值设置限制。

前提条件

您必须已创建工作流、为架构设置了布局并进行了链接并为所有元素定义了 **IN**、**OUT** 和异常绑定。

步骤

1 单击 **Presentation** 选项卡。

   为此工作流定义的两个输入参数已列出。

2 单击 `(VC:VirtualMachine)vm` 参数。

3 在屏幕下半部的 **Description** 选项卡中添加描述。

   例如，键入**要启动的虚拟机。**

4 单击屏幕下半部的 **Properties** 选项卡。

   使用此选项卡可以设置 `(VC:VirtualMachine)vm` 参数的属性。

5 右键单击 **Properties** 选项卡，然后选择 **Add Property**。

6 从建议的属性列表中选择 **Mandatory input**。

   选择此属性时，用户仅在提供要启动的虚拟机之后才能运行 Start VM and Send Email 工作流。

7 将 **Mandatory input** 属性的值设置为 **Yes**。

8 右键单击 **Properties** 选项卡，并再次选择 **Add Property**。

9 从建议的属性列表中选择 **Select value as**。

   设置此属性之后，您即设置了用户选择 `(VC:VirtualMachine)vm` 输入参数值的方式。

10 从可能值的列表中选择 **list**。

11 单击 **Presentation** 选项卡上半部中的 `(string)toAddress` 参数。

12 在屏幕下半部的 **Description** 选项卡中添加描述。

   例如，键入**要通知的人的电子邮件地址。**

13 单击 `(string)toAddress` 的 **Properties** 选项卡。

14 右键单击 **Properties** 选项卡，然后依次选择 **Add Property > Mandatory input**。

15 将 **Mandatory input** 属性的值设置为 **Yes**。

16 右键单击 **Properties** 选项卡，然后依次选择 **Add Property > Matching regular expression**。

   使用此属性，您可以对用户提供的输入内容设置限制。

17 单击 **Matching regular expression** 的 **Value** 文本框，将限制设置为

   `[a-zA-Z0-9_%-+.]+@[a-zA-Z0-9-.]+\.[a-zA-Z]{2,4}`

   设置这些限制可限制用户输入电子邮件地址适用的字符。如果用户在启动工作流时尝试为收件人的电子邮件地址输入任何其他字符，则工作流不会启动。

您已将这两个参数设成为强制性参数，定义了用户可通过什么方式选择要启动的虚拟机，并限制了可对收件人的电子邮件地址输入的字符。

下一步

必须创建输入参数对话框的布局或呈现方式，用户将在运行工作流时在此对话框中输入工作流的输入参数值。

设置简单工作流示例输入参数对话框的布局

您可以在工作流工作台的 **Presentation** 选项卡中创建输入参数对话框的布局或呈现方式。输入参数对话框在用户运行工作流时打开，它是用户用来输入工作流运行所需的输入参数的工具。

在 **Presentation** 选项卡中定义的布局也适用于使用 Web 视图运行的工作流的输入参数对话框。

前提条件

您必须已创建工作流、为架构设置了布局并进行了链接、为所有元素定义了 **IN**、**OUT** 和异常绑定，并且已设置属性和参数属性。

步骤

1 在工作流工作台中单击 **Presentation** 选项卡。

2 右键单击呈现方式层次结构列表中的 **Presentation** 节点，然后选择 **New Group**。

   New Step 节点和 **New Group** 子节点将显示在 **Presentation** 节点下方。

3 右键单击 **New Step**，然后选择 **Delete**。

   因为此工作流仅有两个参数，所以输入参数对话框中不需要多层显示区域。

4 双击 **New Group** 编辑组名称，并按 Enter。

   例如，将显示组命名为 **`Virtual Machine`**。

   用户启动工作流时，您在此处输入的文本将作为标题显示在输入参数对话框中。

5 在 **Presentation** 选项卡底部的 **General** 选项卡中，在 **Description** 文本框中提供 Virtual Machine 显示组的描述。

   例如，键入**选择要启动的虚拟机**。

   用户启动工作流时，您在此处输入的文本将作为提示信息显示在输入参数对话框中。

6 拖动 **Virtual Machine** 显示组下方的 `(VC:VirtualMachine)vm` 参数。

   输入参数对话框（用户在其中输入要启动的虚拟机）中的文本框将显示在“Virtual Machine”标题下方。

7 重复步骤 1 到步骤 6 为 `toAddress` 参数创建显示组，并设置下列属性：

   a 创建名为 **`Recipient's Email Address`** 的显示组。

   b 添加显示组的描述，例如，**输入要通知其此虚拟机已启动的人的电子邮件地址**。

   c 拖动“Recipient's Email Address”显示组下面的 `toAddress` 属性。

您已为用户运行工作流时显示的输入参数对话框设置了布局。

下一步

您已完成简单工作流示例的开发。现在，可以验证和运行该工作流。

验证和运行简单工作流示例

创建工作流之后，可对其进行验证以发现任何可能的错误。如果工作流不包含错误，则可以运行它。

前提条件

在尝试验证和运行工作流之前，您必须已创建工作流、为其设置了架构布局、定义了链接和绑定、定义了参数属性，并且创建了输入参数对话框的呈现方式。

步骤

1 在工作流工作台的 **Schema** 选项卡中，单击 **Validation**。

   验证工具将会找出工作流定义中的所有错误。

2 消除所有错误之后，单击工作流工作台底部的 **Save and Close**。

   返回到 Orchestrator 客户端。

3 单击 **Workflows** 视图。

4 在工作流层次结构列表中依次选择 **Workflow Examples > Start VM and Send Email**。

5 右键单击 Start VM and Send Email 工作流，并选择 **Execute workflow**。

   此时将打开输入参数对话框，并提示您输入要启动的虚拟机和要通知的人员的电子邮件地址。

6 从建议的列表中选择要在 vCenter Server 中启动的虚拟机。

7 输入要向其发送电子邮件通知的人员的电子邮件地址。

8 单击 **Submit** 启动工作流。

   此时，将在 Start VM and Send Email 工作流的下方显示一个工作流令牌。

9 单击工作流令牌以在此工作流运行时跟进它的运行进度。

如果工作流运行成功，则指定的虚拟机将处于开机状态，并且您定义的电子邮件收件人将会收到一封确认电子邮件。

开发复杂工作流

开发复杂示例工作流的过程将向您展示工作流开发过程（以及更高级的方案）中最常见的步骤，如创建自定义判定和循环。

在此练习中，您将开发一个工作流，此工作流会对包含在给定资源池中的所有虚拟机创建快照。所创建的工作流将执行以下任务：

1 提示用户输入包含要对其创建快照的虚拟机的资源池。

2 确定该资源池是否包含正在运行的虚拟机。

3 确定该资源池中包含多少台正在运行的虚拟机。

4 验证池中正在运行的某一台虚拟机是否满足创建快照的特定条件。

5 对该虚拟机创建快照。

6 确定池中是否存在更多要创建快照的虚拟机。

7 重复验证和快照过程，直到工作流对资源池中所有合格的虚拟机都创建了快照。

前提条件

尝试开发此复杂工作流之前，请先完成第 50 页，“开发简单示例工作流”中的练习。本节中的步骤提供了开发过程的主要步骤，但不如简单工作流练习中步骤那么详细。

步骤

1 创建复杂工作流示例第 71 页，

   在此练习中，您将创建一个名为 Take a Snapshot of All Virtual Machines in a Resource Pool 的工作流。

2 定义复杂工作流示例的输入参数第 71 页，

   您可以在工作流工作台中定义工作流输入参数。输入参数可为工作流提供要处理的数据。

3 创建复杂工作流示例的自定义操作第 72 页，

   Check VM 可编脚本元素可调用 Orchestrator API 中不存在的操作。您必须创建 `getVMDiskModes` 操作。

4 创建复杂工作流示例架构第 73 页，

   您可以在工作流工作台的 **Schema** 选项卡上创建工作流的架构。工作流架构包含工作流运行的元素。

5 链接复杂工作流示例架构元素第 74 页，

   您可以在工作流工作台的 **Schema** 选项卡上将工作流的元素链接在一起。链接用于定义工作流的逻辑流。

6 创建复杂工作流示例区域第 75 页，

   您可以选择通过添加工作流注释来突出显示工作流的不同区域。通过创建不同的工作流区域，可使复杂的工作流架构更便于阅读和理解。

7 定义复杂工作流示例的绑定第 76 页，

   您可以在工作流工作台的 **Schema** 选项卡中将工作流的元素绑定在一起。绑定用于定义工作流中数据流动的方式。此外，您还可以将可编脚本任务元素与其 JavaScript 函数绑定。

8 设置复杂工作流示例属性的特性第 85 页，

   您可以在工作流工作台的 **General** 选项卡中设置属性的特性。

9 创建复杂工作流示例输入参数的布局第 85 页，

   您可以在工作流工作台的 **Presentation** 选项卡中创建输入参数对话框的布局或呈现方式。输入参数对话框在用户运行工作流时打开，它是用户用来输入工作流运行所需的输入参数的工具。

10 验证和运行复杂工作流示例第 86 页，

   创建工作流之后，可对其进行验证以发现任何可能的错误。如果工作流不包含错误，则可以运行它。

创建复杂工作流示例

在此练习中，您将创建一个名为 Take a Snapshot of All Virtual Machines in a Resource Pool 的工作流。

前提条件

要创建这个较为复杂的工作流示例，系统中必须安装和配置有以下组件。

- Orchestrator 4.0
- vCenter 4.0，用于控制包含部分虚拟机的资源池
- 工作流层次结构列表中的 **Workflow Examples** 类别，即您在第 51 页，“创建简单工作流示例”中创建的类别。

有关如何安装和配置 vCenter 的详细信息，请参见《ESX 和 vCenter Server 安装指南》。有关如何配置 Orchestrator 的详细信息，请参见《Orchestrator 4.0 管理指南》。

步骤

1 选择 **Workflows > Workflow Examples**。
2 创建一个名为 Take a Snapshot of All Virtual Machines in a Resource Pool 的工作流。
3 通过右键单击新工作流并选择 **Edit** 打开工作流工作台。
4 在工作流工作台的 **General** 选项卡中，单击版本号数字以增大版本号。

   由于这是初次创建的工作流，因此可将版本设置为 **0.0.1**。
5 在 **General** 选项卡中选中 **Allowed operations** 复选框，以设置用户可在此工作流上执行的操作。
6 单击 **General** 选项卡中的 **Server restart behavior** 值，以设置工作流是否在服务器重新启动之后恢复运行。
7 在 **General** 选项卡的 **Description** 文本框中，提供工作流的用途描述。
8 在 **General** 选项卡的底部单击 **Save**。

您已创建 Take a Snapshot of All Virtual Machines in a Resource Pool 工作流。

下一步

现在，您可以继续编辑 Take a Snapshot of All Virtual Machines in a Resource Pool 工作流。

定义复杂工作流示例的输入参数

您可以在工作流工作台中定义工作流输入参数。输入参数可为工作流提供要处理的数据。

前提条件

您必须已创建 Take a Snapshot of All Virtual Machines in a Resource Pool 工作流，并且已在工作流工作台中打开此工作流以对其进行编辑。

步骤

1 在工作流工作台中单击 **Inputs** 选项卡。
2 定义以下输入参数。
   - 名称：`resourcePool`
   - 类型：`VC:ResourcePool`
   - 描述：**包含要创建快照的虚拟机的资源池。**

3 在工作流工作台中单击 **Outputs** 选项卡。

4 定义以下输入参数。

- 名称：`snapshotVmArrayOut`
- 类型：`Array/VC:ResourcePool`
- 描述：**已创建快照的虚拟机阵列。**

您已定义工作流输入参数。

下一步

可以创建一个工作流架构。

创建复杂工作流示例的自定义操作

Check VM 可编脚本元素可调用 Orchestrator API 中不存在的操作。您必须创建 `getVMDiskModes` 操作。

有关如何创建操作的详细信息，请参见第 87 页，第 3 章“开发操作”。

前提条件

必须已创建 Take a Snapshot of All Virtual Machines in a Resource Pool 工作流。

步骤

1 通过单击 **Save and Close**，关闭工作流工作台。

2 在 Orchestrator 客户端中单击 **Actions** 视图。

3 右键单击操作层次结构列表的根目录，然后选择 **New Module**。

4 将新模块命名为 **`com.vmware.example`**。

5 右键单击 **com.vmware.example** 模块，然后选择 **Add Action**。

6 创建名为 `getVMDiskModes` 的操作。

7 右键单击 `getVMDiskModes`，然后选择 **Edit**。

8 通过单击版本数字，在操作工作台的 **General** 选项卡中增大版本号。

9 在 **General** 选项卡中，选中所有 **Allowed Operations** 复选框。

10 在 **General** 选项卡中添加下列操作描述。

```
此操作会返回一个阵列，该阵列包含虚拟机上所有磁盘的磁盘模式。
阵列包含以下字符串值：
- persistent
- independent-persistent
- nonpersistent
- independent-nonpersistent
旧版值：
- undoable
- append
```

11 单击 **Scripting** 选项卡。

12 右键单击 **Scripting** 选项卡的顶部窗格，然后选择 **Add Parameter** 以创建下列输入参数。

- 名称：`vm`
- 值：`VC:VirtualMachine`
- 描述：**要返回磁盘模式的虚拟机**

13 在 **Scripting** 选项卡的底部添加以下脚本。

以下代码会返回虚拟机磁盘的磁盘模式阵列。

```
var devicesArray = vm.config.hardware.device;
var retArray = new Array();
if (devicesArray!=null && devicesArray.length!=0) {
    for (i in devicesArray) {
        if (devicesArray[i] instanceof VcVirtualDisk) {
            retArray.push(devicesArray[i].backing.diskMode);
        }
    }
}
return retArray;
```

14 单击 **Save and Close**，退出 **Actions** 选项板。

您已定义 Take a Snapshot of All Virtual Machines in a Resource Pool 工作流需要的自定义操作。

下一步

必须创建工作流架构。

创建复杂工作流示例架构

您可以在工作流工作台的 **Schema** 选项卡上创建工作流的架构。工作流架构包含工作流运行的元素。

前提条件

您必须已创建 Take a Snapshot of All Virtual Machines in a Resource Pool 工作流、已定义其输入参数并创建了 `getVMDiskModes` 操作。

步骤

1 右键单击 Take a Snapshot of All Virtual Machines in a Resource Pool 工作流以打开工作流工作台。

2 在工作流工作台中单击 **Schema** 选项卡。

3 将以下架构元素添加到工作流架构中。

元素类型	元素名称	架构中的位置
可编脚本任务	Initializing	在开始元素下方
判定	VMs to Process?	在 Initializing 可编脚本任务下方
可编脚本任务	Pool Has No VMs	在 Virtual Machines to Process? 自定义判定下方
自定义判定	Remaining VMs?	在 Virtual Machines to Process? 自定义判定右侧
操作	`getVMDiskModes`	在 Virtual Machines Remaining? 自定义判定右侧
自定义判定	Create snapshot?	在 `getVMDiskModes` 操作右侧
工作流	Create a Snapshot (vCenter Server 4.0)	在 Create snapshot? 判定上方
可编脚本任务	VM Snapshots	在 Create a Snapshot 工作流左侧
可编脚本任务	Increment	在 VM Snapshots 可编脚本任务左侧
可编脚本任务	Log Exception	在 VM Snapshots 可编脚本任务上方

元素类型	元素名称	架构中的位置
可编脚本任务	Set Output	在 Remaining VMs? 自定义判定下方
结束元素	无名称	在 Set Output 可编脚本任务右侧

4 在 **Schema** 选项卡的底部单击 **Save**。

您已创建工作流的架构。

下一步

现在，可以将这些工作流元素链接在一起。

链接复杂工作流示例架构元素

您可以在工作流工作台的 **Schema** 选项卡上将工作流的元素链接在一起。链接用于定义工作流的逻辑流。

前提条件

您必须已创建 Take a Snapshot of All Virtual Machines in a Resource Pool 工作流、已定义其输入参数并创建了相应的架构。

步骤

1 在工作流工作台的 **Schema** 选项卡顶部的工具栏中，单击连接器工具按钮。

2 在架构中的元素之间创建以下链接。

开始元素	目标元素
开始元素	Initializing 脚本元素
Initializing 脚本元素	VMs to Process? 自定义判定
VMs to Process? 判定的 `true` 结果	VMs Remaining? 自定义判定
VMs to Process? 判定的 `false` 结果	Has No VMs 可编脚本任务
Has No VMs 可编脚本任务	Set Output 可编脚本任务
VMs Remaining? 自定义判定的 `true` 结果	`getVMDisksModes` 操作
VMs Remaining? 自定义判定的 `false` 结果	Set Output 可编脚本任务
`getVMDisksModes` 操作	Create Snapshot? 判定
`getVMDisksModes` 操作异常链接	Log Exception 可编脚本任务
Create Snapshot? 自定义判定的 `true` 结果	Create a Snapshot 工作流
Create Snapshot? 自定义判定的 `false` 结果	Increment 可编脚本任务
Create a Snapshot 工作流	VM Snapshots 可编脚本任务
Create a Snapshot 工作流异常链接	Log Exception 可编脚本任务
VM Snapshots 可编脚本任务	Increment 可编脚本任务
Increment 可编脚本任务	VMs Remaining? 自定义判定
Log Exception 可编脚本任务	Increment 可编脚本任务
Set Output 可编脚本任务	结束元素

3 在工作流工作台的 **Schema** 选项卡底部单击 **Save**。

您已将工作流元素链接在一起，从而定义了工作流的逻辑流。

下一步

您可以选择通过使用工作流注释定义多个工作流区域。

创建复杂工作流示例区域

您可以选择通过添加工作流注释来突出显示工作流的不同区域。通过创建不同的工作流区域，可使复杂的工作流架构更便于阅读和理解。

前提条件

您必须已创建工作流及其架构，并且已链接架构元素。

步骤

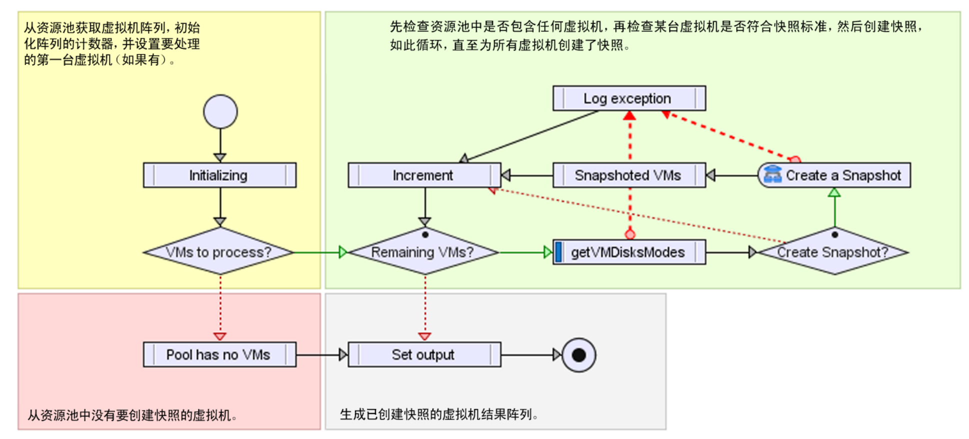
1 通过使用工作流注释，创建以下工作流区域。

区域中的元素	描述
开始元素；Initialize 可编脚本任务；VMs to Process? 自定义判定	**从资源池获取虚拟机阵列，初始化阵列的计数器，并设置要处理的第一台虚拟机（如果有）。**
Pool has no VMs 可编脚本任务。	**资源池中没有要创建快照的虚拟机。**
VMs remaining? 自定义判定；`getVMDisksModes` 操作，Create Snapshot? 判定；Create a Snapshot 工作流；VM Snapshots 可编脚本任务；Increment 可编脚本任务；Log Exception 可编脚本任务	**先检查资源池中是否包含任何虚拟机，再检查某台虚拟机是否符合快照标准，然后创建快照，如此循环，直至为所有虚拟机创建了快照。**
Set Output 可编脚本任务；结束元素	**生成已创建快照的虚拟机结果阵列。**

2 在工作流工作台的 **Schema** 选项卡底部单击 **Save**。

您的工作流架构应如下图所示。

图 2–4 Take Snapshot of all Virtual Machines in a Resource Pool 示例工作流的架构图



下一步

您必须定义元素参数之间的绑定。

定义复杂工作流示例的绑定

您可以在工作流工作台的 **Schema** 选项卡中将工作流的元素绑定在一起。绑定用于定义工作流中数据流动的方式。此外，您还可以将可编脚本任务元素与其 JavaScript 函数绑定。

前提条件

您必须已创建 Take a Snapshot of All Virtual Machines in a Resource Pool 工作流，并且定义了相应的输入参数、创建了架构并已将架构元素链接在一起。

步骤

1 在工作流工作台中单击 **Schema** 选项卡。

2 定义第 76 页，“复杂工作流示例绑定”中显示的绑定。

3 在工作流工作台的 **Schema** 选项卡底部单击 **Save**。

此时，所有元素的输入和输出参数已与相应的参数类型和值绑定。

下一步

必须设置属性的特性。

复杂工作流示例绑定

绑定可定义简单工作流示例的操作元素处理输入和输出参数的方式。

Take Snapshots of All Virtual Machines in a Resource Pool 工作流需要以下输入和输出参数绑定。您还将为可编脚本任务元素定义 JavaScript 函数。

Initializing 可编脚本任务

Initializing 可编脚本任务元素可初始化工作流的属性。表 2‑14 显示了 Initializing 可编脚本任务元素需要的输入和输出参数绑定。

表 2‑14 Initializing 可编脚本任务元素的绑定

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
resourcePool	IN	绑定	■ 本地参数：resourcePool ■ 源参数：resourcePool[in-parameter] ■ 类型：VC:ResourcePool ■ 描述：**包含要创建快照的虚拟机的资源池**
allVMs	OUT	创建	■ 本地参数：allVMs ■ 源参数：allVMs[attribute] ■ 类型：Array/VC:VirtualMachine ■ 描述：**资源池中的虚拟机。**

表 2–14 Initializing 可编脚本任务元素的绑定（续）

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
numberOfVMs	OUT	创建	■ 本地参数：numberOfVMs ■ 源参数：numberOfVMs[attribute] ■ 类型：数字 ■ 描述：**在 resourcePool 中找到的虚拟机的数量**
vmCounter	OUT	创建	■ 本地参数：vmCounter ■ 源参数：vmCounter[attribute] ■ 类型：数字 ■ 描述：**阵列内部虚拟机的计数器**
vm	OUT	创建	■ 本地参数：vm ■ 源参数：vm[attribute] ■ 类型：VC:VirtualMachine ■ 描述：**当前已创建快照的虚拟机**
snapshotVmArray	OUT	创建	■ 本地参数：snapshotVmArray ■ 源参数：snapshotVmArray[attribute] ■ 类型：Array/VC:VirtualMachine ■ 描述：**已创建快照的虚拟机阵列**

Initialize 可编脚本任务元素执行以下脚本函数。

```
//Retrieve an array of virtual machines contained in the specified Resource Pool
allVMs = resourcePool.vm;
//Initialize the size of the Array and the first VM to snapshot
if (allVMs!=null && allVMs.length!=0) {
    numberOfVms = allVMs.length;
    vm = allVMs[0];
} else {
    numberOfVms = 0;
}
//Initialize the VM counter
vmCounter = 0;
//Initializing the array of VM snapshots
snapshotVmArray = new Array();
```

VMs to Process? 判定元素

VMs to Process? 判定元素可确定资源池中是否存在要对其创建快照的任何虚拟机。表 2–15 显示了 VMs to Process? 判定元素需要的绑定。

表 2–15 VMs to Process? 判定元素的绑定

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
numberOfVMs	判定	绑定	■ 源参数：numberOfVMs[attribute] ■ 判定语句：Greater than ■ 值：0.0 ■ 描述：**在 resourcePool 中找到的虚拟机的数量**

Pool Has No VMs 可编脚本任务元素

Pool Has No VMs 可编脚本任务元素可记录 Orchestrator 数据库的资源池中不包含合格虚拟机的事实。表 2–16 显示了 Pool Has No VMs 可编脚本任务元素需要的绑定。

表 2–16 Pool Has No VMs 可编脚本任务元素的绑定

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
resourcePool	IN	绑定	■ 本地参数：resourcePool ■ 源参数：resourcePool[in-parameter] ■ 类型：VC:ResourcePool ■ 描述：**包含要创建快照的虚拟机的资源池。**

Pool Has No VMs 可编脚本任务元素执行以下脚本函数。

```
//Writes the following event in the vCO database
Server.warn("The specified ResourcePool "+resourcePool.name+" does not contain any VMs.");
```

Remaining VMs? 自定义判定元素

Remaining VMs? 自定义判定元素可确定任何要创建快照的虚拟机是否仍保留在资源池中。表 2–17 显示了 Remaining VMs? 自定义判定元素需要的绑定。

表 2–17 Remaining VMs? 自定义判定元素的绑定

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
numberOfVMs	IN	绑定	■ 源参数：numberOfVMs[attribute] ■ 判定语句：Greater than ■ 值：0.0 ■ 描述：**在 resourcePool 中找到的虚拟机的数量**
vmCounter	IN	绑定	■ 本地参数：vmCounter ■ 源参数：vmCounter[attribute] ■ 类型：数字 ■ 描述：**阵列内部虚拟机的计数器**

Remaining VMs? 自定义判定元素执行以下脚本函数。

```
//Checks if the workflow has reached the end of the array of VMs
if (vmCounter < numberOfVms) {
    return true;
} else {
    return false;
}
```

getVMDisksModes 操作元素

getVMDisksModes 操作元素可获取虚拟机中运行的磁盘的模式。表 2–18 显示了 getVMDisksModes 操作元素需要的绑定。

表 2–18 getVMDisksModes 操作元素的绑定

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
vm	IN	绑定	■ 本地参数：vm ■ 源参数：vm[attribute] ■ 类型：VC:VirtualMachine ■ 描述：**当前已创建快照的虚拟机**
actionResult	OUT	创建	■ 本地参数：actionResult ■ 源参数：vmDisksModes[attribute] ■ 类型：数组/字符串 ■ 描述：**虚拟机当前的磁盘模式**
errorCode	异常	创建	本地参数：errorCode

Create Snapshot? 自定义判定元素

Create Snapshot? 自定义判定元素可确定是否对虚拟机创建快照，具体取决于虚拟机的磁盘模式。表 2–19 显示了 Create Snapshot? 自定义判定元素需要的绑定。

表 2–19 Create Snapshot? 判定元素的绑定

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
vmDisksMode	IN	绑定	■ 本地参数：vmDisksMode ■ 源参数：vmDisksMode[attribute] ■ 类型：数组/字符串 ■ 描述：**虚拟机当前的磁盘模式**
vm	IN	绑定	■ 本地参数：vm ■ 源参数：vm[attribute] ■ 类型：VC:VirtualMachine ■ 描述：**当前已创建快照的虚拟机**

Create Snapshot? 自定义判定元素执行以下脚本函数。

```
//A snapshot cannot be taken if one of its disks is in independent mode
// (independent-persistent or independent-nonpersistent)
var containsIndependentDisks = false;
if (vmDisksModes!=null && vmDisksModes.length>0) {
    for (i in vmDisksModes) {
        if (vmDisksModes[i].charAt(0)=="i") {
            containsIndependentDisks = true;
        }
    }
} else {
    //if no disk found no need to try to snapshot the VM
    System.warn("Won't snapshot '"+vm.name+"', no disks found");
    return false;
}
if (containsIndependentDisks) {
    System.warn("Won't snapshot '"+vm.name+"', independent disk(s) found");
    return false;
} else {
    System.log("Snapshoting '"+vm.name+"'");
    return true;
}
```

Create Snapshot 工作流元素

Create Snapshot 工作流元素可对虚拟机创建快照。表 2–20 显示了 Create Snapshot 工作流元素需要的绑定。

表 2–20 Create Snapshot 工作流元素的绑定

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
vm	IN	绑定	■ 本地参数：vm ■ 源参数：ActiveVM[attribute] ■ 类型：VC:VirtualMachine ■ 描述：**要创建快照的活动虚拟机。**
name	IN	创建	■ 本地参数：name ■ 源参数：snapshotName[attribute] ■ 类型：字符串 ■ 描述：**此快照的名称。对于此虚拟机，该名称不需要具有唯一性。**
description	IN	创建	■ 本地参数：description ■ 源参数：snapshotDescription[attribute] ■ 类型：字符串 ■ 描述：**此快照的描述。**
memory	IN	创建	■ 本地参数：memory ■ 源参数：snapshotMemory[attribute] ■ 类型：布尔 ■ 值：no ■ 描述：**如果为 TRUE，则快照中包括虚拟机内部状态的转储（内存转储）。**
quiesce	IN	创建	■ 本地参数：quiesce ■ 源参数：snapshotQuiesce[attribute] ■ 类型：布尔 ■ 值：yes ■ 描述：**如果为 TRUE，并且虚拟机在创建快照时已启动，则 VMware Tools 用于静默虚拟机中的文件系统。**
snapshot	OUT	创建	■ 本地参数：snapshot ■ 源参数：NULL ■ 类型：VC:VirtualMachineSnapshot ■ 描述：**已创建的快照。**
errorCode	异常	创建	本地参数：errorCode

VM Snapshots 可编脚本任务元素

VM Snapshots 可编脚本任务元素可将快照添加到阵列中。表 2-21 显示了 VM Snapshots 可编脚本任务元素需要的绑定。

表 2-21 VM Snapshots 可编脚本任务元素的绑定

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
vm	IN	绑定	■ 本地参数：vm ■ 源参数：ActiveVM[attribute] ■ 类型：VC:VirtualMachine ■ 描述：**要创建快照的活动虚拟机。**
snapshotVmArray	IN	绑定	■ 本地参数：snapshotVmArray ■ 源参数：snapshotVmArray[attribute] ■ 类型：Array/VC:VirtualMachine ■ 描述：**已创建快照的虚拟机阵列**
snapshotVmArray	OUT	绑定	■ 本地参数：snapshotVmArray ■ 源参数：snapshotVmArray[attribute] ■ 类型：Array/VC:VirtualMachine ■ 描述：**已创建快照的虚拟机阵列**

VM Snapshots 可编脚本任务元素执行以下脚本函数。

```
//Writes the following event in the vCO database
Server.log("Successfully took snapshot of the VM '"+vm.name);
//Inserts the VM snapshot in an array
snapshotVmArray.push(vm);
```

Increment 可编脚本任务元素

Increment 可编脚本任务元素可增加用于计算阵列中虚拟机数量的计数器。表 2-22 显示了 Increment 可编脚本任务元素需要的绑定。

表 2–22 Increment 可编脚本任务元素的绑定

参数名称	绑定类型	与现有参数绑定还是创建参数？	绑定值
vmCounter	IN	绑定	■ 本地参数：vmCounter ■ 源参数：vmCounter[attribute] ■ 类型：数字 ■ 描述：**阵列内部虚拟机的计数器**
allVMs	IN	绑定	■ 本地参数：allVMs ■ 源参数：allVMs[attribute] ■ 类型：Array/VC:VirtualMachine ■ 描述：**资源池中的虚拟机。**
vmCounter	OUT	绑定	■ 本地参数：vmCounter ■ 源参数：vmCounter[attribute] ■ 类型：数字 ■ 描述：**阵列内部虚拟机的计数器**
vm	OUT	绑定	■ 本地参数：vm ■ 源参数：vm[attribute] ■ 类型：VC:VirtualMachine ■ 描述：**当前已创建快照的虚拟机**

Increment 可编脚本任务元素执行以下脚本函数。

```
//Increases the array VM counter
vmCounter++;
//Sets the next VM to be snapshot in the attribute vm
vm = allVMs[vmCounter];
```

Log Exception 可编脚本任务元素

Log Exception 可编脚本任务元素可处理工作流和操作元素中的异常。表 2–23 显示了 Log Exception 可编脚本任务元素需要的绑定。

表 2–23 Log Exception 任务元素的绑定

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
vm	IN	绑定	■ 本地参数：vm ■ 源参数：vm[attribute] ■ 类型：VC:VirtualMachine ■ 描述：**当前已创建快照的虚拟机**
errorCode	IN	绑定	■ 本地参数：errorCode ■ 源参数：errorCode[attribute] ■ 类型：字符串 ■ 描述：**对虚拟机创建快照时捕获的异常**

Log Exception 可编脚本任务元素执行以下脚本函数。

```
//Writes  the following event in the vCO database
Server.error("Coudln't snapshot the VM '"+vm.name+"', exception:"+errorCode);
```

Set Output 可编脚本任务元素

Set Output 可编脚本元素可生成工作流的输出参数，此参数包含已对其创建快照的虚拟机阵列。表 2–24 显示了 Set Output 可编脚本任务元素需要的绑定。

表 2–24 Set Output 任务元素的绑定

参数名称	绑定类型	与现有参数绑定还是创建参数?	绑定值
snapshotVmArray	IN	绑定	■ 本地参数：snapshotVmArray ■ 源参数：snapshotVmArray[attribute] ■ 类型：Array/VC:VirtualMachine ■ 描述：**已创建快照的虚拟机阵列**
snapshotVmArrayOut	OUT	绑定	■ 本地参数：snapshotVmArrayOut ■ 源参数：snapshotVmArrayOut[out-parameter] ■ 类型：Array/VC:VirtualMachine ■ 描述：**已创建快照的虚拟机的阵列**

Set Output 可编脚本任务元素执行以下脚本函数。

```
//Passes the value of the internal attribute to a workflow output parameter
snapshotVmArrayOut = snapshotVmArray;
```

设置复杂工作流示例属性的特性

您可以在工作流工作台的 **General** 选项卡中设置属性的特性。

前提条件

您必须已创建工作流，已创建并链接了架构，并且已为所有元素定义了 IN 和 OUT 绑定。

步骤

1. 单击 **General** 选项卡。
2. 选中以下属性的只读复选框以将这些属性设置为只读常量：
   - `snapshotName`
   - `snapshotDescription`
   - `snapshotMemory`
   - `snapshotQuiesce`

您已定义哪些工作流属性为常量，哪些为变量。

下一步

您必须创建工作流呈现方式，用于创建输入参数对话框的布局，用户可在运行工作流时在此对话框中输入工作流的输入参数值。

创建复杂工作流示例输入参数的布局

您可以在工作流工作台的 **Presentation** 选项卡中创建输入参数对话框的布局或呈现方式。输入参数对话框在用户运行工作流时打开，它是用户用来输入工作流运行所需的输入参数的工具。

前提条件

您必须已创建工作流，已创建和链接了相应的架构，为所有元素定义了 IN、OUT 和异常绑定，并且已设置属性和参数属性。

步骤

1. 在工作流工作台中单击 **Presentation** 选项卡。

   此工作流只有一个输入参数，因此创建呈现方式比较简单。
2. 右键单击呈现方式层次结构列表中的 **Presentation** 节点，然后选择 **New Group**。
3. 删除在 **New Group** 上方显示的 **New Step**。
4. 双击 **New Group**，并将组名称更改为 **`Resource Pool`**。
5. 在 **Presentation** 选项卡底部的 **General** 选项卡中，在 **Description** 文本框中提供 Resource Pool 显示组的描述。

   例如，**输入包含要创建快照的虚拟机的资源池名称。**
6. 单击 `(VC:ResourcePool)resourcePool` 参数。
7. 单击 `(VC:ResourcePool)resourcePool` 的 **Properties** 选项卡。
8. 右键单击 **Properties** 选项卡，然后依次选择 **Add Property > Mandatory input**。

9 再次右键单击 **Properties** 选项卡，并从建议的属性列表中选择 **Select value as**。

   设置此属性之后，您即设置了用户选择 `(VC:ResourcePool)resourcePool` 输入参数值的方式。

10 拖动 **Resource Pool** 显示组下面的 `(VC:ResourcePool)resourcePool` 参数。

您已为用户运行工作流时显示的输入参数对话框创建了布局。

下一步

您已完成较复杂工作流示例的开发。现在，可以验证和运行该工作流。

验证和运行复杂工作流示例

创建工作流之后，可对其进行验证以发现任何可能的错误。如果工作流不包含错误，则可以运行它。

前提条件

在尝试验证和运行工作流之前，您必须已创建工作流、为其设置了架构布局、定义了链接和绑定、定义了参数属性，并且创建了输入参数对话框的呈现方式。

步骤

1 在工作流工作台的“Schema”选项卡中，单击 **Validation**。

   验证工具可检测到工作流定义中存在的任何错误。

2 消除所有错误之后，单击工作流工作台底部的 **Save and Close**。

   返回到 Orchestrator 客户端。

3 单击 **Workflows** 视图。

4 在工作流层次结构列表中，依次选择 **Workflow Examples > Take a Snapshot of All Virtual Machines in a Resource Pool**。

5 右键单击 Take a Snapshot of All Virtual Machines in a Resource Pool 工作流，然后选择 **Execute**。

   此时将打开输入参数对话框，提示您输入包含要对其创建快照的虚拟机的资源池。

6 单击 **Submit** 以运行工作流。

   此时，将在 Take a Snapshot of All Virtual Machines in a Resource Pool 工作流的下方显示一个工作流令牌。

7 单击工作流令牌以在此工作流运行时跟进它的运行进度。

如果工作流运行成功，则该工作流会对所选资源池中的所有虚拟机创建快照。

开发操作 3

Orchestrator 提供多个预定义的操作库。操作代表您在工作流、Web 视图和脚本中用作构建块的各个函数。

操作是 JavaScript 函数。它们包含多个输入参数以及一个返回值。它们可以在 Orchestrator API 中的任何对象上调用，也可以在通过插件导入到 Orchestrator 中的任何 API 中的对象上调用。

当工作流运行时，操作会从该工作流的属性中获取输入参数。这些属性可以是工作流的初始输入参数，也可以是工作流中其他元素在其运行时设置的属性。

Orchestrator 客户端使用操作图标 () 来标识操作。

本章讨论了以下主题：

重用操作

在将一个函数定义为操作而不是将其直接编码为可编脚本任务工作流元素时，将在库中公开它。当操作出现在库中时，其他工作流即可使用它。

如果独立于调用操作的工作流单独定义操作，即可更轻松地更新或优化这些操作。此外，还可以通过定义单个操作来使其他工作流重用操作。工作流运行时，Orchestrator 仅在工作流首次运行各个操作时缓存操作。Orchestrator 然后即可重用缓存中的操作。缓存操作对工作流中的递归调用或快速循环非常有用。

可以复制操作，将其导出到其他工作流或软件包，或将其移到操作层次结构列表中的其他类别下。

访问“Actions”视图

Orchestrator 客户端界面上具有一个 **Actions** 视图，可用于访问 Orchestrator 服务器的操作库。

在位于 Orchestrator 客户端界面左侧的 **Actions** 视图中，包含一个在 Orchestrator 服务器中可用的所有操作的层次结构列表。

步骤

1 单击客户端界面左侧的 **Actions**。

2 通过展开操作层次结构列表的节点，浏览操作库。

使用 **Actions** 视图，可以查看库中的操作的相关信息，此外还可以创建和编辑操作。

“Actions”视图的组件

在操作层次结构列表中单击某个操作时，此操作的相关信息将出现在 Orchestrator 客户端的右窗格中。

Actions 视图包含四个选项卡。

General	显示有关操作的常规信息，包括名称、版本号、权限和描述。
Scripting	显示操作的返回类型、输入参数和用于定义操作函数的 JavaScript 代码。
Events	显示此操作遇到或触发的所有事件。
Permissions	显示哪些用户和用户组有权访问此操作。

创建操作

可以将单个函数定义为诸如工作流之类的其他元素可以使用的操作。操作就是包含指定的输入和输出参数和权限的 JavaScript 函数。

- 创建操作第 88 页，

  将一个函数定义为操作而不是将其直接编码为可编脚本任务工作流元素时，可在库中公开该操作以供其他工作流使用。

- 查找执行操作的元素第 89 页，

  如果对操作进行编辑时更改了它的行为，则有可能会不慎中断执行该操作的工作流或应用程序。Orchestrator 提供了一项功能，可查找执行某个给定元素的所有操作、工作流或软件包。您可以检查修改元素是否会影响其他元素的操作。

- 操作编码准则第 89 页，

  要想优化工作流性能并尽可能重用操作，创建操作时应遵从基本编码准则。

创建操作

将一个函数定义为操作而不是将其直接编码为可编脚本任务工作流元素时，可在库中公开该操作以供其他工作流使用。

步骤

1 在 Orchestrator 客户端中单击 **Actions** 视图。

2 展开操作层次结构列表的根节点，并导航到您要在其中创建操作的模块。

3 右键单击该模块并选择 **Add action**。

4 在文本框中输入操作的名称，然后单击 **OK**。

5 右键单击操作并选择 **Edit**。

6 单击 **Scripting** 选项卡。

7 要更改默认的返回类型，请单击 **void** 链接。

8 通过单击箭头图标添加操作的输入参数。

9 编写操作脚本。

10 单击 **Save and close**。

您的自定义操作即会添加到操作库中。

下一步

此时便可以在工作流中使用新的自定义操作了。

查找执行操作的元素

如果对操作进行编辑时更改了它的行为，则有可能会不慎中断执行该操作的工作流或应用程序。Orchestrator 提供了一项功能，可查找执行某个给定元素的所有操作、工作流或软件包。您可以检查修改元素是否会影响其他元素的操作。

重要事项 **Find Elements that Use this Element** 功能会检查所有软件包、工作流和策略，但不会检查脚本。因此，修改操作可能会影响在 **Find Elements that Use this Element** 功能无法识别的某个脚本中调用此操作的元素。

步骤

1 在 Orchestrator 客户端中单击 **Actions** 视图。

2 展开操作层次结构列表的节点以导航到给定的操作。

3 右键单击该操作并选择 **Find Elements that Use this Element**。

   此时将打开一个对话框，其中显示执行此操作的所有元素，如工作流或软件包。

4 双击结果列表中的某个元素以在 Orchestrator 客户端中显示该元素。

您已找到执行给定操作的所有元素。

下一步

可以检查修改此元素是否会影响其他任何元素。

操作编码准则

要想优化工作流性能并尽可能重用操作，创建操作时应遵从基本编码准则。

基本操作准则

创建操作时，必须使用基本准则。

- 每个操作必须包含一段角色和功能的描述。
- 应编写简短的基本操作，然后再将它们组合为一个工作流。
- 避免编写执行多项功能的操作，因为这将限制操作的重用可能性。
- 避免编写长期运行的操作。而应在工作流中创建循环，并在操作元素之后包括一个等待事件或等待定时器元素。
- 不要在操作中编写检查点。工作流会在每个元素的运行起点和终点设置检查点。
- 避免在操作中编写循环，而应在工作流中创建循环。如果服务器重新启动，则正在运行的工作流会在元素启动时在最后一个检查点恢复。如果您在操作中写入了一个循环，并且当工作流正在运行该操作时服务器重新启动，工作流会在该操作开头的检查点处恢复，循环则再次重新开始。

操作命名准则

命名操作时，请使用基本准则。

- 使用英语书写操作名称。
- 操作名称以小写字母开头。对名称中的每个联合单词使用大写字母开头。例如，`myAction`。
- 使操作名称尽可能清楚明了，以便明确指出操作的功能。例如，`backupAllVMsInPool`。

- 使模块名称尽可能清楚明了。
- 使模块名称唯一。
- 对模块名称使用反序的互联网地址格式。例如，`com.vmware.myactions.myAction`。

操作参数和属性准则

请在编写操作参数和属性定义时使用基本准则。

- 使用英语书写参数和属性的名称。
- 参数和属性名称以小写字母开头。
- 参数和属性名称要尽可能清楚明了。
- 最好将参数和属性名称限定为一个单词。如果名称必须包含多个单词，则对名称中的每个联合单词使用大写字母开头。例如，`myParameter`。
- 对表示一个对象数组的参数和属性使用复数形式。
- 使变量名称含义明确，例如，`displayName`。
- 在每个属性中包含一段描述以说明其用途。
- 不要在一个操作中使用过多参数。

脚本 4

Orchestrator 使用 JavaScript 创建可用于创建操作、工作流元素和策略的构建块。

Orchestrator 使用 Mozilla Rhino JavaScript 引擎来提供一种编写新脚本的方式。该脚本引擎可提供版本控制、变量类型检查、命名空间管理、自动完成和异常处理等功能。

Mozilla Rhino JavaScript 引擎仅使用基本 JavaScript 语言功能，如 if 条件、循环、数组和字符串。可以在脚本中使用的其他对象包括 Orchestrator API 提供的对象，或者您通过插件导入到 Orchestrator 中以及映射到 JavaScript 对象的任何其他 API 中的对象。

本章讨论了以下主题：

需要编写脚本的 Orchestrator 元素

并非所有 Orchestrator 元素都需要您编写脚本。为了使应用程序具备最大的灵活性，可以通过添加 JavaScript 函数自定义某些元素。

可以在以下 Orchestrator 元素中添加脚本。

操作
操作是一种脚本函数。理想情况下，应将为操作编写的脚本限制为单个操作，从而最大化其他元素（如其他工作流）重用操作的可能性。

策略
您使用监视触发器事件的脚本来设置策略。策略会在触发器事件发生时启动定义的耦合操作。

工作流
Scriptable Task 工作流元素可用于编写自定义脚本操作，或者是可以在工作流中使用的一系列操作。此外，您也可以使用返回 `true` 或 `false` 的脚本来定义自定义判定元素的布尔判定语句。

使用 Orchestrator API

Orchestrator API 可作为 JavaScript 对象以及 Orchestrator 可通过其插件访问的所有对象的方法公开。

例如，可以通过 Orchestrator API 访问 vCenter Server 4.0 API 的 JavaScript 实现，以将其用于所创建的脚本元素。此外，您还可以从 Orchestrator 服务器中安装的其他插件来访问对象的 JavaScript 实现。创建第三方应用程序的插件时，您会将对象从其 API 映射到 Orchestrator API 随后公开的 JavaScript 对象。

可以使用在 Orchestrator 中实施的 Mozilla Rhino JavaScript 引擎功能来帮助您编写脚本。

步骤

1 从工作流工作台访问脚本引擎第 92 页，

Orchestrator 脚本引擎实施 Mozilla Rhino JavaScript 引擎，可帮助您为工作流中的脚本元素编写脚本。从工作流工作台中的 **Scripting** 选项卡访问脚本工作流元素的脚本引擎。

2 从操作或策略工作台访问脚本引擎第 93 页，

Orchestrator 脚本引擎实施 Mozilla Rhino JavaScript 引擎，可帮助您编写操作或策略的脚本。可以从操作和策略工作台的 **Scripting** 选项卡访问操作和策略的脚本引擎。

3 访问 Orchestrator API Explorer 第 93 页，

Orchestrator 提供 API Explorer，您可以通过它搜索 Orchestrator API 并将 JavaScript 对象添加到脚本元素中。

4 使用 Orchestrator API Explorer 查找对象第 93 页，

Orchestrator API 公开了所有插件技术的 API，包括整个 vCenter Server API。Orchestrator API Explorer 可帮助您查找需要添加到脚本中的对象。

5 将 JavaScript 对象添加到脚本第 94 页，

编写脚本时，Orchestrator 脚本引擎可帮助您添加对象和函数。自动插入对象和函数以及自动完成脚本行均可加快脚本处理速度，此外还能将脚本中发生编写错误的可能性降至最低。

6 将参数添加到脚本中第 95 页，

Orchestrator 脚本引擎可帮助您将可用参数导入到脚本中。

7 使用 JavaScript 访问 Java 类第 95 页，

默认情况下，Orchestrator 会限制 JavaScript 只能访问一组有限的 Java 类。如果您需要 JavaScript 能够访问更大范围的 Java 类，则必须设置一个 Orchestrator 系统属性以授予此访问权限。

从工作流工作台访问脚本引擎

Orchestrator 脚本引擎实施 Mozilla Rhino JavaScript 引擎，可帮助您为工作流中的脚本元素编写脚本。从工作流工作台中的 **Scripting** 选项卡访问脚本工作流元素的脚本引擎。

步骤

1 在 Orchestrator 客户端的 **Workflows** 视图中右键单击某个工作流，然后选择 **Edit**。

2 在工作流工作台中单击 **Schema** 选项卡。

3 将 Scriptable Task 元素或 Custom Decision 元素添加到工作流架构中。

4 单击可编写脚本的元素的 **Scripting** 选项卡。

您访问了脚本引擎，以便定义工作流元素的脚本函数。通过 **Scripting** 选项卡，您可以在 API 之间导航、参阅有关对象的文档、搜索对象以及编写 JavaScript 脚本。

下一步

使用 API Explorer 搜索 Orchestrator API。

从操作或策略工作台访问脚本引擎

Orchestrator 脚本引擎实施 Mozilla Rhino JavaScript 引擎，可帮助您编写操作或策略的脚本。可以从操作和策略工作台的 **Scripting** 选项卡访问操作和策略的脚本引擎。

步骤

1 在 Orchestrator 客户端的 **Actions** 或 **Policies** 视图中右键单击某个操作或策略，然后选择 **Edit**。

2 在操作或策略工作台中单击 **Scripting** 选项卡。

您访问了脚本引擎，以便定义操作或策略元素的脚本函数。通过 **Scripting** 选项卡，您可以在 API 之间导航、参阅有关对象的文档、搜索对象以及编写 JavaScript 脚本。

下一步

使用 API Explorer 搜索 Orchestrator API。

访问 Orchestrator API Explorer

Orchestrator 提供 API Explorer，您可以通过它搜索 Orchestrator API 并将 JavaScript 对象添加到脚本元素中。

步骤

◆ 您可以从 Orchestrator 客户端，或者工作流、策略和操作工作台的 **Scripting** 选项卡访问 API Explorer。

- 要从 Orchestrator 客户端访问 API Explorer，请在 Orchestrator 客户端工具栏中单击 **Tools > API Explorer**。
- 要从工作流、策略和操作工作台的 **Scripting** 选项卡访问 API Explorer，请单击这些工作台左侧的 **Search API**。

此时将显示 API Explorer，可用于搜索 Orchestrator API 的所有对象和函数。

下一步

使用 API Explorer 为可编写脚本的元素编写脚本。

使用 Orchestrator API Explorer 查找对象

Orchestrator API 公开了所有插件技术的 API，包括整个 vCenter Server API。Orchestrator API Explorer 可帮助您查找需要添加到脚本中的对象。

前提条件

API Explorer 已打开。

步骤

1 在 API Explorer 的 **Search** 文本框中输入对象的名称或部分名称，然后单击 **Search**。

   要将搜索限制为某个特定对象类型，请取消选中或选中 **Scripting Class**、**Attributes & Methods** 和 **Types & Enumerations** 复选框。

2 双击建议列表中的元素。

   该对象会在左侧的层次结构列表中突出显示。层次结构列表下方的文档窗格将显示此对象的相关信息。

您已找到要查找的对象。

下一步

在脚本中使用找到的对象。

API Explorer 中的 JavaScript 对象类型

API Explorer 在 **Scripting** 选项卡左侧的层次结构树中或 API Explorer 对话框中标识不同类型的 JavaScript 对象，并将它们组合在一起。API Explorer 使用不同的图标来帮助您标识不同的对象类型。

表 4-1 介绍 Orchestrator API 的不同对象类型，并显示相关图标。

表 4–1 Orchestrator API 中的对象类型

对象类型	层次结构列表中的图标	描述
类型		类型
函数集		包含一组静态方法的内部类型
原始		原始类型
对象		标准 Orchestrator 脚本对象
属性		JavaScript 属性
方法		JavaScript 方法
构造函数	C	JavaScript 构造函数
枚举		JavaScript 枚举
字符串集		字符串集，默认值
模块		操作类别
插件	由插件定义的图像	通过插件向 Orchestrator 公开的 API

将 JavaScript 对象添加到脚本

编写脚本时，Orchestrator 脚本引擎可帮助您添加对象和函数。自动插入对象和函数以及自动完成脚本行均可加快脚本处理速度，此外还能将脚本中发生编写错误的可能性降至最低。

前提条件

要编辑的脚本元素已打开，并且该元素的 **Scripting** 选项卡也已打开。

步骤

1 在 **Scripting** 选项卡左侧的对象层次结构列表中导航，或使用 API Explorer 搜索函数，以选择要添加到脚本中的对象。

2 右键单击对象并选择 **Copy**。

   如果脚本引擎不允许您复制某个对象，该对象不会出现在脚本的上下文中。

3 右键单击脚本编写面板，然后将复制的对象粘贴到脚本中的相应位置。

   脚本引擎会将该对象输入到脚本中，并为其提供构造函数和实例名称以完成该脚本。

   例如，如果复制 `Date` 对象，则脚本引擎会将以下代码粘贴到脚本中。

```
var myDate = new Date();
```

4 复制并粘贴某种方法以添加到脚本中。

   脚本引擎通过添加所需的属性完成方法调用。

   例如，如果从 `com.vmware.library.vc.vm` 模块复制了 `cloneVM()` 方法，脚本引擎会将以下代码粘贴到脚本中。

```
System.getModule("com.vmware.library.vc.vm").cloneVM(vm,folder,name,spec)
```

   脚本引擎会突出显示已在元素中定义的参数。所有未定义的参数都将以普通方式显示。

5 将光标置于您粘贴到脚本中的对象的结尾处，然后按 Ctrl+空格键，以便从此对象可调用的可能方法和属性的上下文相关列表中进行选择。

   注意 自动完成功能目前尚处于实验阶段。

6 继续在左侧的层次结构列表中执行复制和粘贴操作，直至将所有所需对象和函数都添加到脚本中。

您已将对象和函数添加到脚本中。

下一步

将参数添加到脚本中。

将参数添加到脚本中

Orchestrator 脚本引擎可帮助您将可用参数导入到脚本中。

如果已经为要编辑的元素定义了参数，则它们将以链接形式显示在 **Scripting** 选项卡工具栏中。

前提条件

要编辑的脚本元素已打开，并且该元素的 **Scripting** 选项卡也已打开。

步骤

1 在 **Scripting** 选项卡的脚本编写面板中，将光标移至脚本中的适当位置。

2 在 **Scripting** 选项卡工具栏中，单击参数链接。

   Orchestrator 会将此参数插入光标所在的位置。

3 将具有 Null 值的参数插入到脚本中。

   如果将 Null 值传递给原始类型（如整数、布尔和字符串），Orchestrator 脚本 API 会自动为此参数设置默认值。

您已将参数添加到脚本中。

下一步

在脚本中添加 Java 类的访问权限。

使用 JavaScript 访问 Java 类

默认情况下，Orchestrator 会限制 JavaScript 只能访问一组有限的 Java 类。如果您需要 JavaScript 能够访问更大范围的 Java 类，则必须设置一个 Orchestrator 系统属性以授予此访问权限。

允许 JavaScript 引擎对 Java 虚拟机 (JVM) 具有完全访问权限会导致潜在的安全问题。有缺陷或恶意的脚本可能有权访问运行 Orchestrator 服务器的用户能够访问的所有系统组件。因此，默认情况下，Orchestrator JavaScript 引擎只能访问 `java.util.*` 软件包中的类。

如果您需要 JavaScript 对除 `java.util.*` 软件包以外的类具有访问权限，可在一个配置文件中列出要对其授予 JavaScript 访问权限的 Java 软件包。然后，将 `com.vmware.scripting.rhino-class-shutter-file` 系统属性设置为指向此文件。

步骤

1 创建一个文本配置文件以存储要对其授予 JavaScript 访问权限的 Java 软件包列表。

例如，要授予 JavaScript 对 `java.net` 软件包中的所有类以及 `java.lang.Object` 类的访问权限，请将以下内容添加到文件中。

```
java.net.*
java.lang.Object
```

2 使用适当的名称将配置文件保存到适当的位置。

3 打开 `vmo.properties` 系统属性文件。

`vmo.properties` 文件位于以下位置：

- *<安装-目录>*`\VMware\Orchestrator\app-server\server\vmo\conf`（如果安装的是 Orchestrator 独立版本）。
- *<安装-目录>*`\VMware\Infrastructure\Orchestrator\app-server\server\vmo\conf`（如果使用 vCenter Server 安装程序安装 Orchestrator）。

4 通过将以下行添加到 `vmo.properties` 文件中，设置 `com.vmware.scripting.rhino-class-shutter-file` 系统属性。

`com.vmware.scripting.rhino-class-shutter-file=`*您的配置文件的路径*

5 保存 `vmo.properties` 文件。

6 重新启动 Orchestrator 服务器。

JavaScript 引擎将对您指定的 Java 类拥有访问权限。

异常处理准则

Orchestrator 中的 Mozilla Rhina JavaScript 引擎实施支持异常处理，可用于处理发生的错误。使用脚本编写异常处理程序时，必须遵循以下准则。

- 请使用下列欧洲计算机制造商协会 (ECMA) 错误类型。使用 `Error` 作为插入函数返回的一般异常，并使用以下特定错误类型。
  - `TypeError`
  - `RangeError`
  - `EvalError`
  - `ReferenceError`
  - `URIError`
  - `SyntaxError`

  以下示例将显示 `URIError` 定义。

```
try {
     ...
throw new URIError("VirtualMachine with ID 'vm-0056'
not found on 'vcenter-test-1'") ;
```

```
    ...
} catch ( e if e instanceof URIError ) {

}
```

- 避免在脚本中创建本地错误类型，而应使用预定义的错误类型。请不要在脚本中使用以下类型的错误处理。

```
try {
    ...
throw new MyError("this is a message") ;
    ...
} catch ( e if e instanceof MyError ) {

}
```

- 脚本不捕获的所有异常必须是 <类型>:SPACE<人可读的消息> 形式的简单字符串对象，如以下示例所示。

```
throw "ValidationError:The input parameter 'myParam' of type 'string' is too short."
```

- 将人可读的消息编写得尽可能清楚明白。
- 简单字符串异常类型检查必须使用以下模式。

```
try {
throw "VMwareNoSpaceLeftOnDatastore:Datastore 'myDatastore' has no space left" ;
} catch ( e if (typeof(e)=="string" && e.indexOf("VMwareNoSpaceLeftOnDatastore:") == 0) ) {
System.log("No space left on device") ;
// Do something useful here
}
```

- 简单字符串异常类型检查必须在工作流的脚本元素中使用以下模式。

```
if (typeof(errorCode)=="string"
&& errorCode.indexOf("VMwareNoSpaceLeftOnDatastore:")
      == 0) {
// Do something useful here
}
```

Orchestrator JavaScript 示例

您可以剪切、粘贴和改写 Orchestrator JavaScript 示例，来编写常见耦合任务的 JavaScript 脚本。

- 基本脚本示例第 98 页，

  工作流脚本元素、操作和策略都需要编写常见任务的基本脚本。您可以剪切或改写这些示例，然后将其粘贴到您的脚本元素中。

- 电子邮件脚本示例第 100 页，

  工作流脚本元素、操作和策略都需要编写常见的电子邮件相关任务的脚本。您可以剪切或改写这些示例，然后将其粘贴到您的脚本元素中。

- 文件系统脚本示例第 100 页，

  工作流脚本元素、操作和策略都需要编写常见文件系统任务的脚本。您可以剪切或改写这些示例，然后将其粘贴到您的脚本元素中。

- LDAP 脚本示例第 101 页，

  工作流脚本元素、操作和策略都需要编写常见 LDAP 任务的脚本。您可以剪切或改写这些示例，然后将其粘贴到您的脚本元素中。

- 日志记录脚本示例第 101 页，

  工作流脚本元素、操作和策略都需要编写常见日志记录任务的脚本。您可以剪切或改写这些示例，然后将其粘贴到您的脚本元素中。

- 网络脚本示例第 101 页，

  工作流脚本元素、操作和策略都需要编写常见网络任务的脚本。您可以剪切或改写这些示例，然后将其粘贴到您的脚本元素中。

- vCenter Server 脚本示例第 102 页，

  工作流脚本元素、操作和策略都需要编写常见 vCenter Server 任务的脚本。您可以剪切或改写这些示例，然后将其粘贴到您的脚本元素中。

- 工作流脚本示例第 104 页，

  工作流脚本元素、操作和策略都需要编写常见工作流任务的脚本。您可以剪切或改写这些示例，然后将其粘贴到您的脚本元素中。

基本脚本示例

工作流脚本元素、操作和策略都需要编写常见任务的基本脚本。您可以剪切或改写这些示例，然后将其粘贴到您的脚本元素中。

检查是否存在变量

以下 JavaScript 示例将检查工作流中是否存在变量。

```
function isDefined(variable) {
    try {
        eval(""+variable+"") ;
        return true ;
    } catch ( ex) {
        return false ;
    }
}
var a = 10 ;
if ( isDefined('a') ) {
    System.log("is define") ;
} else {
    System.log("Not defined") ;
}
```

从哈希表设置和获取属性

以下 JavaScript 示例将在哈希表中设置属性，同时还从哈希表中获取属性。在以下示例中，关键字始终是字符串，而值可为对象、数字、布尔值或字符串。

```
var table = new Properties() ;
table.put("myKey",new Date()) ;
// get the object back
var myDate= table.get("myKey") ;
System.log("Date is :"+myDate) ;
```

替换字符串的内容

以下 JavaScript 示例将使用新内容替换字符串的现有内容。

```
var str1 = "'hello'" ;
var reg = new RegExp("(')", "g");
var str2 = str1.replace(reg,"\\'") ;
System.log(""+str2) ; // result :\'hello\'
```

访问 XML 文档

以下 JavaScript 示例允许您通过使用 ECMAScript for XML (E4X) 从 JavaScript 访问 XML 文档。

```
var people = <people>
<person id="1">
<name>Moe</name>
</person>
<person id="2">
<name>Larry</name>
</person>
</people>;
System.log("Native XML datatype :" + typeof(people));
System.log("which is a list :" + people.person.length());
System.log("whose elements are indexable :" + people.person[0]);
people.person[0].@id='47';
System.log("and mutable!: " + people.person.(name=='Moe'));
delete people.person[0];
people.person[1] = new XML("<person id=\"3\"><name>James</name></person>");
for each(var person in people..person){
    System.log("- " + person.name);
}
```

比较类型

以下 JavaScript 示例将检查对象是否与给定对象类型匹配。

```
var path = 'myurl/test';
if(typeof(path, String)){
throw("string");
else {
throw("other");
}
```

在 Orchestrator 服务器中运行命令

以下 JavaScript 示例可用于在 Orchestrator 服务器上运行命令行。请使用启动服务器时使用的凭据。

```
var cmd = new Command("ls -al") ;
cmd.execute(true) ;
System.log(cmd.output) ;
```

电子邮件脚本示例

工作流脚本元素、操作和策略都需要编写常见的电子邮件相关任务的脚本。您可以剪切或改写这些示例，然后将其粘贴到您的脚本元素中。

获取电子邮件地址

以下 JavaScript 示例将获取正在运行的脚本当前所有者的电子邮件地址。

```
var emailAddress = Server.getRunningUser().emailAddress ;
```

发送电子邮件

以下 JavaScript 示例可将包含指定内容的电子邮件通过 SMTP 服务器发送给指定的收件人。

```
var message = new EmailMessage() ;
message.smtpHost = "smtpHost" ;
message.subject= "my subject" ;
message.toAddress = "receiver@vmware.com" ;
message.fromAddress = "sender@vmware.com" ;
message.addMimePart("This is a simple message","text/html") ;
message.sendMessage() ;
```

文件系统脚本示例

工作流脚本元素、操作和策略都需要编写常见文件系统任务的脚本。您可以剪切或改写这些示例，然后将其粘贴到您的脚本元素中。

获取文件的内容

以下 JavaScript 示例将从 Orchestrator 服务器主机获取文件的内容。

```
var fileReader = new FileReader("/home/vmware/readme.txt") ;
fileReader.open() ;
var fileContentAsString = fileReader.readAll();
fileReader.close() ;
```

将内容添加到简单文本文件

以下 JavaScript 示例可将内容添加到文本文件中。

```
var fileWriter = new FileWriter("/home/vmware/readme.txt") ;
fileWriter.open() ;
fileWriter.writeLine("File written at :"+new Date()) ;
fileWriter.writeLine("Another line") ;
fileWriter.close() ;
```

LDAP 脚本示例

工作流脚本元素、操作和策略都需要编写常见 LDAP 任务的脚本。您可以剪切或改写这些示例，然后将其粘贴到您的脚本元素中。

将 LDAP 对象转换为 Active Directory 对象

以下 JavaScript 示例将 LDAP 组元素转换为 Active Directory 用户组对象，此外还进行了反向转换。

```
var ldapGroup ;
// convert from ldap element to Microsoft:UserGroup object
var adGroup = ActiveDirectory.search("UserGroup",ldapGroup.commonName) ;
// convert back to LdapGroup element
var ldapElement = Server.getLdapElement(adGroup.distinguishedName) ;
```

日志记录脚本示例

工作流脚本元素、操作和策略都需要编写常见日志记录任务的脚本。您可以剪切或改写这些示例，然后将其粘贴到您的脚本元素中。

永久日志记录

以下 JavaScript 示例将创建永久的日志条目。

```
Server.log("This is a persistant message", "enter a long description here");
Server.warn("This is a persistant warning", "enter a long description here");
Server.error("This is a persistant error", "enter a long description here");
```

非永久日志记录

以下 JavaScript 示例将创建非永久日志条目。

```
System.log("This is a non-persistant log message");
System.warn("This is a non-persistant log warning");
System.error("This is a non-persistant log error");
```

网络脚本示例

工作流脚本元素、操作和策略都需要编写常见网络任务的脚本。您可以剪切或改写这些示例，然后将其粘贴到您的脚本元素中。

从 URL 获取文本

以下 JavaScript 示例将访问某个 URL，获取文本，并将其转换为字符串。

```
var url = new URL("http://www.vmware.com") ;
var htmlContentAsString = url.getContent() ;
```

vCenter Server 脚本示例

工作流脚本元素、操作和策略都需要编写常见 vCenter Server 任务的脚本。您可以剪切或改写这些示例，然后将其粘贴到您的脚本元素中。

访问受管对象类型

以下 JavaScript 示例使 Orchestrator 可以通过 vCenter Server 4.0 插件使用脚本访问 vCenter Server 受管对象。

```
var vm = ...;
// Get the property 'name'
var name = vm.name;   // returns a string
// return a VcEnvironmentBrowser managed object
var environmentBrowser = vm.environmentBrowser;
```

访问数据对象类型

以下 JavaScript 示例使 Orchestrator 可以通过 vCenter Server 4.0 插件使用脚本访问 vCenter Server 数据对象。

```
var vimHost = ...// vCenter Server host connection
var virtualMachineSnapshotInfo = ...// VcVirtualMachineSnapshotInfo data object

// There no automatic conversion between ManagedObjectReference and VimManagedObject
// in a 'Data Object Type'. virtualMachineSnapshotRef is only the reference to the
// 'Managed Object Type' not the object itself
var virtualMachineSnapshotRef = virtualMachineSnapshotInfo.currentSnapshot;

// Convert from ManagedObjectReference to a VimManagedObject.
// The concrete class is VcVirtualMachineSnapshot.
var virtualMachineSnapshot = VcPlugin.convertToVimManagedObject(vimHost,
virtualMachineSnapshotRef);

// The reverse operation, if required.
// virtualMachineSnapshotInfo.currentSnapshot = virtualMachineSnapshot.reference;
```

处理枚举类型

以下 JavaScript 示例使 Orchestrator 可以通过 vCenter Server 4.0 插件使用脚本处理 vCenter Server 枚举。

```
// a VcSharesLevel FINDER ENUMERATION TYPE, for example
// received from an input parameter
var sharesLevel = ...

// get the String value of the FINDER ENUMERATION TYPE
var sharesLevelString = sharesLevel.name;

// Convert from the String value to a static value of VcSharesLevel
// SCRIPTING TYPE
var level = VcSharesLevel.fromString(sharesLevel.name);

// Get the String value of the VcSharesLevel SCRIPTING TYPE
System.log("Shares Level :" + level.value);

// Get a FINDER ENUMERATION
finder = Server.findForType("VC:SharesLevel", "normal");
```

查找主机和虚拟机

以下 JavaScript 示例使 Orchestrator 可以通过 vCenter Server 4.0 插件使用脚本查找主机和虚拟机。

```
var vimHosts = VcPlugin.getVimHosts();
System.log(vimHosts.length + " Vim hosts found");
for (var i = 0; i < vimHosts.length; i++) {
    var vimHost = vimHosts[i];
    System.log("Vim host '" + vimHost.id + "'");

    // Hierarchy entry point
    var rootFolder = vimHost.rootFolder;

    // Get the property 'name'
    var name = rootFolder.name;
    System.log("--- Root folder '" + name + "'");

    // Get the folder's data centers
    var datacenters = rootFolder.datacenter;
    if  (datacenters != null) {
        for (var j = 0; j < datacenters.length; j++) {
            var datacenter = datacenters[j];
            System.log("--- Datacenter '" + datacenter.id + "'");
        }
    }

    // Method to get all the host systems in a vCenter Server host
    var hostSystems = vimHost.getAllHostSystems();
    if  (hostSystems != null) {
        for (var j = 0; j < hostSystems.length; j++) {
            var hostSystem = hostSystems[j];
            System.log("--- HostSystem '" + hostSystem.id + "'");
        }
    }

    // Method to get all the virtual machines in a vCenter Server host
    var vms = vimHost.getAllVirtualMachines();
    if (vms != null) {
        for (var j = 0; j < vms.length; j++) {
            var vm = vms[j];
            System.log("--- VM '" + vm.id + "'");
            System.log("--- VM '" + vm.getName() + "'");
            var guestInfo = vm.guest;
            System.log("--- VM guestInfo '" + guestInfo + "'");
            if (guestInfo != null) {
        System.log("--- VM guestInfo.guestFamily '" + guestInfo.guestFamily + "'");
            }
        }
    }
}
```

工作流脚本示例

工作流脚本元素、操作和策略都需要编写常见工作流任务的脚本。您可以剪切或改写这些示例，然后将其粘贴到您的脚本元素中。

返回当前用户运行的所有工作流

以下 JavaScript 示例将从服务器获取所有工作流运行，然后检查这些工作流是否属于当前用户。例如，可以在 Webview 组件中使用此脚本。

```
var allTokens = Server.findAllForType('WorkflowToken');
var currentUser = Server.getCredential().username;
var res = [];
for(var i = 0; i<res.length; i++){
    if(allTokens[i].runningUserName == currentUser){
      res.push(allTokens[i]);
    }
}
return res;
```

调度工作流

以下 JavaScript 示例将启动包含给定属性集的工作流，然后设置一小时后启动的调度计划。

```
var workflowToLaunch = myWorkflow ;
// create parameters
var workflowParameters = new Properties() ;
workflowParameters.put("name","John Doe") ;
// change the task name
workflowParameters.put("__taskName","Workflow for John Doe") ;

// create scheduling date one hour in the future
var workflowScheduleDate = new Date() ;
var time = workflowScheduleDate.getTime() + (60*60*1000) ;
workflowScheduleDate.setTime(time) ; var scheduledTask =
workflowToLaunch.schedule(workflowParameters,workflowScheduleDate);
```

创建软件包 5

软件包是将内容从一台 Orchestrator 服务器传输到另一台服务器时所使用的载体。软件包可以包含工作流、操作、策略、Web 视图、配置或资源。

将一个元素添加到软件包中时，Orchestrator 会检查其依赖性并将所有相关元素都添加到软件包中。例如，如果添加了一个使用多步操作或其他工作流的工作流，Orchestrator 会将这些操作和工作流添加到软件包中。

当您导入软件包时，服务器会将其中所包含的不同元素的版本与相匹配的本地元素相比较。比较结果将显示本地元素和导入的元素之间的版本差异。管理员可以决定是导入该软件包还是选择导入某些特定元素。

软件包通过数字权限管理功能来控制接收服务器使用软件包内容的方式。Orchestrator 会给软件包签名并对其进行加密以便保护其数据。通过使用 X509 证书，软件包可以跟踪哪些用户导出和重新分配了元素。

重要事项 Orchestrator 3.2 生成的软件包可向上兼容 Orchestrator 4.0。可以将 Orchestrator 3.2 服务器中的软件包导入到 Orchestrator 4.0 服务器中。Orchestrator 4.0 中的软件包不向后兼容 Orchestrator 3.2。无法将由 Orchestrator 4.0 服务器生成的软件包导入到 Orchestrator 3.2 服务器中。

创建软件包

您可以采用软件包方式导出工作流、策略、操作、插件、资源、Web 视图和配置元素。

前提条件

您必须具有要添加到软件包中的元素（如工作流、操作和策略）。

步骤

1 在 Orchestrator 客户端中单击 **Packages** 视图。

2 在 **Packages** 层次结构列表的标题栏中单击菜单按钮，然后选择 **Add package**。

3 在打开的对话框中提供软件包名称，然后单击 **OK**。

   软件包的命名约定为 `<域.贵公司>.category.<软件包名称>`。例如，`com.vmware.mycategory.mypackage`。

4 右键单击软件包并选择 **Edit**。

   此时将打开软件包编辑器。

5 在 **General** 选项卡中添加该软件包的描述。

6 单击 **Workflows** 选项卡以将工作流添加到软件包中。

   - 单击 **Insert Workflows (list search)** 以搜索并在选取对话框中选择所需的工作流。
   - 单击 **Insert Workflows (tree browsing)** 以浏览并在层次结构列表中选择所需的工作流。

7 单击 **Policies**、**Actions**、**Configurations** 和 **Resources** 选项卡以将策略模板、操作、配置元素和资源元素添加到软件包中。

8 在 **Web View** 选项卡中单击 **Insert Webview** 以将 Web 视图添加到软件包中。

9 在 **Used plug-ins** 选项卡中单击 **Insert used plug-in** 以将插件添加到软件包中。

您已将所需的元素添加到软件包。

下一步

您必须为此软件包设置用户权限。

对软件包设置用户权限

您可以对软件包设置不同级别的权限，以便限制不同用户或用户组对该软件包的访问。

从 Orchestrator LDAP 服务器的用户和用户组中选择要为其设置权限的不同用户和用户组。Orchestrator 定义了可对用户或用户组应用的权限级别。

查看	用户可以查看软件包中的元素，但无法查看架构或脚本。
检查	用户可以查看软件包中的元素，包括架构和脚本。
执行	用户可以运行软件包中的元素。
编辑	用户可以编辑软件包中的元素。
管理	用户可以对软件包中的元素设置权限。

前提条件

您必须创建一个软件包，并将其打开以便在软件包编辑器中编辑，此外还必须为其添加必要的元素。

步骤

1 在软件包编辑器中单击 **Permissions** 选项卡。

2 单击 **Add access rights** 链接为新用户或用户组定义权限。

3 通过在 **Search** 文本框中输入文本来搜索用户或用户组。

   搜索结果会显示 Orchestrator LDAP 服务器中与搜索条件相匹配的所有用户和用户组。

4 选择用户或用户组并单击 **OK**。

5 右键单击用户并选择 **Add access rights**。

6 选中相应的复选框为此用户设置权限级别，然后单击 **OK**。

   权限级别不具叠加性。要授予用户查看元素、检查架构和脚本、运行和编辑元素以及更改权限的权限，您必须选中所有的复选框。

7 单击 **Save and Close** 退出软件包编辑器。

您已创建软件包并设置了适当的用户权限。

开发插件 6

通过插件，您可以使用 Orchestrator 访问和控制外部技术和应用程序。可以通过使用插件访问的外部技术包括虚拟化管理工具、电子邮件系统、数据库、目录服务和远程控制接口。

Orchestrator 提供一组标准插件，可用于将诸如 VMware vCenter Server API 和电子邮件功能之类的技术整合到工作流中。此外，Orchestrator 的开放式插件架构允许您开发用于访问其他应用程序的插件。Orchestrator 实施开放标准，可以简化与外部系统的集成。

每个插件均向 Orchestrator 平台公开一个外部产品 API。这些插件可提供对象清单、通过新的对象类型扩展脚本引擎以及向 Orchestrator 触发器发布来自外部系统的通知事件。每个插件还会提供一个或多个软件包，其中包含代表集成产品典型自动化用例的元素。

通过 Orchestrator 开放式插件架构和相关软件开发工具包 (SDK)，您可以打开用于耦合的第三方产品和工具。

本章讨论了以下主题：

插件的组件和架构

Orchestrator 插件必须包含一组标准组件，而且必须遵循标准架构。遵循这些惯例，有助于为尽可能多的各种外部技术创建插件。

- 插件组件第 108 页，

  插件由一组标准组件组成，这些组件向 Orchestrator 平台公开插件技术中的对象。通过这些组件，您可以对插件技术中的对象执行耦合操作。

- 访问 Orchestrator 插件 API 第 109 页，

  Orchestrator 插件 API 可提供为创建插件适配器和插件工厂而实现的 Java 接口。插件适配器和插件工厂向 Orchestrator 服务器公开插件技术的对象和操作。

- 插件适配器的作用第 109 页，

  插件适配器是插件进入 Orchestrator 服务器的入口点。要创建插件适配器，必须通过 Orchestrator 插件 API 实现和扩展 `IPluginAdaptor` 接口。

- 插件工厂的作用第 110 页，

  插件工厂用于定义 Orchestrator 如何查找插件技术中的对象以及如何对这些对象执行操作。要创建插件工厂，必须通过 Orchestrator 插件 API 实现和扩展 `IPluginFactory` 接口。

- 在 vso.xml 文件中定义应用程序映射第 110 页，

  要创建插件，必须定义 Orchestrator 如何访问插件技术中的对象以及如何与这些对象交互。这些交互可在 vso.xml 文件中定义。

- vso.xml 插件定义文件的格式第 111 页，

  vso.xml 文件可定义 Orchestrator 服务器与插件技术交互的方式。必须包含对每种类型的对象或操作的引用，才能在 vso.xml 文件中向 Orchestrator 公开这些对象和操作。

- vso.xml 插件定义文件的元素第 111 页，

  vso.xml 文件包含一组标准元素。有部分元素是必选的，而其他元素是可选的。

- 命名插件对象第 123 页，

  必须为插件在插件技术中找到的每个对象提供一个唯一标识符。可以在 vso.xml 文件和脚本对象的 <finder> 元素中定义对象名称。

- *.dar 文件的结构第 124 页，

  您通过标准 Java 归档 (JAR) 或 ZIP 文件形式交付已完成的插件，但必须将其重命名为 *.dar。然后，将此 *.dar 文件导入到 Orchestrator 服务器中。

插件组件

插件由一组标准组件组成，这些组件向 Orchestrator 平台公开插件技术中的对象。通过这些组件，您可以对插件技术中的对象执行耦合操作。

插件必须具有标准组件的功能。

适配器
用于定义插件技术和 Orchestrator 服务器之间的接口。适配器是插件进入耦合平台的入口点。适配器可创建插件工厂，管理插件的加载和卸载，并管理在插件技术中的对象上发生的事件。

工厂
用于定义 Orchestrator 如何查找插件技术中的对象以及如何对这些对象执行操作。适配器为在 Orchestrator 和插件技术之间打开的每个客户端会话创建工厂。

模块
插件自身，由一个 Java 类集合、一个 vso.xml 文件和一个 Orchestrator 工作流和操作软件包（用于与通过插件访问的对象交互）定义而成。

查找程序
用于定义 Orchestrator 查找和呈现插件技术中对象的方式的交互规则。查找程序在 vso.xml 文件中定义。

清单
Orchestrator 通过查找程序访问的对象可出现在 Orchestrator 客户端的 **Inventory** 视图中。通过定义脚本对象，可以对清单中的对象执行操作。

脚本对象
可用于访问插件技术中的对象、操作和属性的 JavaScript 对象类型。脚本对象用于定义 Orchestrator 通过 JavaScript 访问插件技术的对象模型的方式。您可以访问 Orchestrator 脚本 API 中的脚本对象，然后将它们集成到 Orchestrator 脚本任务、操作和策略中。

事件
监控 Orchestrator 策略通过查找程序找到的插件技术中给定对象的状态。Orchestrator 可定义两类可通过插件监控的策略事件：触发器和量表。

触发器
如果插件技术中发生定义的事件，则会通过插件启动某个特定操作。

监视程序
代表长时间运行的工作流中的 Waiting Event 元素，监视将在插件技术中完成的某个特定事件。您采用可编脚本任务元素定义监视程序监视的触发事件，并将相关输出传递到 Waiting Event 元素。

您在插件适配器和工厂实现中定义这些组件中的所有对象。然后，再将在适配器和工厂实现中定义的对象和操作映射到名为 vso.xml 的 XML 定义文件的 Orchestrator 对象中。

访问 Orchestrator 插件 API

Orchestrator 插件 API 可提供为创建插件适配器和插件工厂而实现的 Java 接口。插件适配器和插件工厂向 Orchestrator 服务器公开插件技术的对象和操作。

- 插件适配器由 `IPluginAdaptor` 接口定义。
- 插件工厂由 `IPluginFactory` 接口定义。

插件 API 包括可在创建适配器和工厂实现时调用的其他接口、类和注释。要查看完整的 Orchestrator 插件 API 类列表，请参见第 140 页，“Orchestrator 插件 API 参考”。

查找插件 API Java 归档文件

Orchestrator 在 Orchestrator 插件 API Java 归档 (JAR) 文件 `vmware-vmo-sdkapi.jar` 中提供插件 API 的类。要开发插件适配器和插件工厂实现，必须在类路径中包括 `vmware-vmo-sdkapi.jar` 文件。

可以在以下位置中查找 `vmware-vmo-sdkapi.jar` 文件：

- *<安装-目录>*`\VMware\Orchestrator\app-server\server\vmo\lib`（如果安装的是 Orchestrator 独立版本）。
- *<安装-目录>*`\VMware\Infrastructure\Orchestrator\app-server\server\vmo\lib`（如果使用 vCenter Server 安装程序安装 Orchestrator）。

导入插件 API 软件包

编写实现 Orchestrator 插件 API 的 Java 类时，必须使用 Java `import` 语句导入以下软件包。

```
import ch.dunes.vso.sdk.api.*;
```

插件适配器的作用

插件适配器是插件进入 Orchestrator 服务器的入口点。要创建插件适配器，必须通过 Orchestrator 插件 API 实现和扩展 `IPluginAdaptor` 接口。

要创建插件适配器，需创建用于实现 `IPluginAdaptor` 接口的 Java 类。您创建的插件适配器类可管理插件技术中的插件工厂、事件和触发器。`IPluginAdaptor` 接口提供可用于执行这些任务的方法。

插件适配器可起到以下主要作用。

创建工厂

插件适配器最重要的作用就是创建插件工厂。插件适配器类可调用 `IPluginAdaptor.createPluginFactory` 方法，从而创建一个用于实现 `IPluginFactory` 接口的类实例。

管理事件

插件适配器是 Orchestrator 服务器和插件技术之间的接口。插件适配器负责管理 Orchestrator 对插件技术中的对象执行或监视的事件。适配器通过事件发布程序管理事件。事件发布程序是适配器通过调用 `IPluginAdaptor.registerEventPublisher` 方法创建的 `IPluginEventPublisher` 接口的实例。事件发布程序对插件技术中的对象设置触发器和量表，这使得 Orchestrator 策略可以在对象上发生某些特定事件或对象的值传递某些特定阈值时启动已定义的操作。同样，您也可以定义 `PluginTrigger` 和 `PluginWatcher` 实例，这两个实例用于定义长时间运行的工作流中 Wait Event 元素等待的事件。

获取插件名称

您在 `vso.xml` 文件中提供插件的名称。插件适配器从 `vso.xml` 文件获取此名称，并将其发布在 Orchestrator 客户端的 **Inventory** 视图中。

有关插件 API 的 `IPluginAdaptor` 接口及其所有方法和所有其他类的详细信息，请参见第 140 页，“Orchestrator 插件 API 参考”。

插件工厂的作用

插件工厂用于定义 Orchestrator 如何查找插件技术中的对象以及如何对这些对象执行操作。要创建插件工厂，必须通过 Orchestrator 插件 API 实现和扩展 `IPluginFactory` 接口。

要创建插件工厂，需创建用于实现 `IPluginFactory` 接口的 Java 类。您创建的插件工厂类可定义 Orchestrator 用于通过插件访问对象的查找程序规则。该工厂允许 Orchestrator 服务器按对象 ID、对象与其他对象的关系或通过搜索查询字符串来查找对象。

插件工厂可起到以下主要作用。

查找对象

您可以创建对象查找程序，按照相关名称和类型查找对象。可使用 `IPluginFactory.find` 方法按名称和类型查找对象。

查找与其他对象相关的对象

您可以创建对象查找程序，通过给定的关系类型查找与给定对象相关的对象。关系在 `vso.xml` 文件中定义。此外，还可以创建查找程序，通过给定关系类型查找与所有父对象相关的子对象。实现 `IPluginFactory.findRelation` 方法可通过给定关系类型查找任何与给定父对象相关的对象。实现 `IPluginFactory.hasChildrenInRelation` 方法可发现父实例是否至少存在一个子对象。

定义查询以按照自己的标准查找对象

可以创建实施您自己选择的查询规则的对象查找程序。实现 `IPluginFactory.findAll` 方法可在工厂调用此方法时查找满足您定义的查询规则的所有对象。您在 `QueryResult` 对象中获取 `findAll` 方法的结果，此对象包含所有找到的与您定义的查询规则匹配的对象列表。

在对象上运行命令

通过实现 `IPluginFactory.executePluginCommand` 方法，可以创建用于运行自定义命令的函数。您将定义要在工厂调用此方法时运行的命令。

有关插件 API 的 `IPluginFactory` 接口及其所有方法和所有其他类的详细信息，请参见第 140 页，“Orchestrator 插件 API 参考”。

在 vso.xml 文件中定义应用程序映射

要创建插件，必须定义 Orchestrator 如何访问插件技术中的对象以及如何与这些对象交互。这些交互可在 `vso.xml` 文件中定义。

`vso.xml` 文件向 Orchestrator 服务器提供以下信息：

- 插件的版本、名称和描述
- 一个或多个数据源，以及相关的插件适配器 Java 类名称
- Orchestrator 服务器启动时插件的行为
- 映射到插件技术中的 Java 类的查找程序
- 在 Orchestrator 脚本 API 中映射到插件技术中的 Java 对象和操作的 JavaScript 对象类型
- 用于定义 ID、名称和描述值列表的枚举
- Orchestrator 策略监控的事件

`vso.xml` 文件必须遵循 Orchestrator 插件 XML 架构定义。可以在 VMware 支持站点访问此架构定义。

`http://www.vmware.com/support/orchestrator/plugin-4-0.xsd`

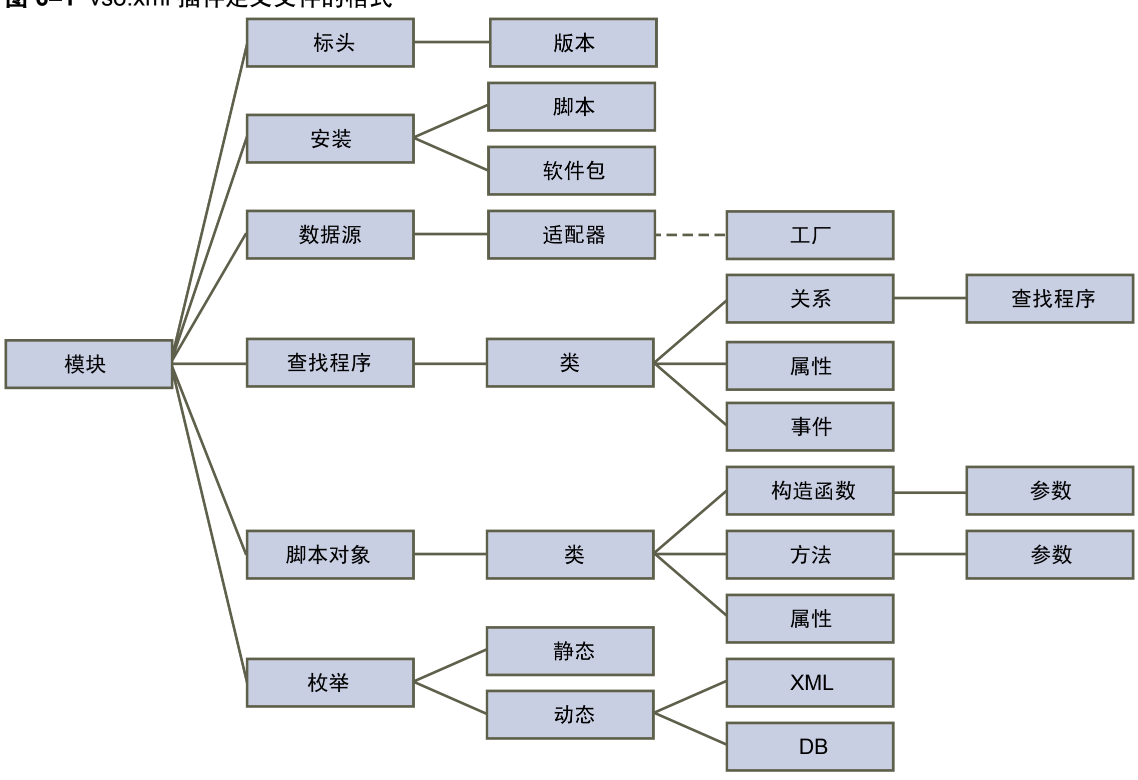
vso.xml 插件定义文件的格式

vso.xml 文件可定义 Orchestrator 服务器与插件技术交互的方式。必须包含对每种类型的对象或操作的引用，才能在 vso.xml 文件中向 Orchestrator 公开这些对象和操作。

作为插件的开放式架构和标准化实现的一部分，vso.xml 文件必须遵循标准格式。

图 6-1 显示 vso.xml 插件定义文件的格式，以及元素相互嵌套的方式。

图 6–1 vso.xml 插件定义文件的格式



vso.xml 插件定义文件的元素

vso.xml 文件包含一组标准元素。有部分元素是必选的，而其他元素是可选的。

每个元素都具有一些属性，可用于定义您映射到 Orchestrator 对象和操作的对象和操作值。此部分内容介绍了 vso.xml 文件的所有元素以及每个元素的可能属性值。

<module> 元素

模块用于描述一组插件对象，以便使其对 Orchestrator 可用。

模块包含有关如何将插件技术中的数据映射到 Java 类、版本管理、如何部署模块以及插件如何出现在 Orchestrator 清单中的信息。

<module> 元素具有以下属性：

属性	值	描述
name	字符串	定义插件中所有 <finder> 元素的类型
version	数字	插件版本号，在新版本的插件中重新加载软件包时使用
build-number	数字	插件的内部版本号，在新版本的插件中重新加载软件包时使用
image	图像文件	显示在 Orchestrator 清单中的图标

属性	值	描述
display-name	字符串	显示在 Orchestrator 清单中的名称
interface-mapping-allowed	true 或 false	VMware 强烈建议不要使用接口映射

父元素 无

子元素

- <description>
- <installation>
- <configuration>
- <webview-component-library>
- <finder-datasources>
- <inventory>
- <finders>
- <scripting-objects>
- <enumerations>

<description> 元素

<description> 元素提供插件的描述。

<description> 元素没有属性或子元素。

<installation> 元素

<installation> 元素使您可以在服务器启动时安装软件包或运行脚本。

<installation> 元素具有以下属性：

属性	值	描述
mode	always、never 或 version	在 Orchestrator 服务器启动时，设置 mode 值会导致以下行为： ■ 操作始终运行 ■ 操作从不运行 ■ 服务器检测到更新版本的插件时操作运行

父元素 <module>

子元素 <action>

<action> 元素

<action> 元素指定 Orchestrator 服务器启动时运行的操作。

<action> 元素属性可提供用于定义插件启动时的行为的 Orchestrator 软件包或脚本的路径。<action> 元素具有以下属性。

属性	值	描述
resource	字符串	从 *.dar 文件的根目录到 Java 软件包或脚本的路径。
type	install-package 或 execute-script	在 Orchestrator 服务器中安装指定的 Orchestrator 软件包，或运行指定的脚本。

父元素 <installation>

子元素 无

<webview-component-library> 元素

<webview-component-library> 元素指向包含自定义 Web 视图 Tapestry 组件的 JAR 文件，这些组件用于扩展 Web 视图的功能。

<webview-component-library> 元素具有以下属性。

属性	值	描述
jar	字符串	包含 Web 视图组件的 JAR 文件
specification-path	字符串	JAR 文件中指向 Tapestry 组件定义文件的路径，该文件位于 *.dar 文件的 lib 文件夹中。路径必须以正斜线 (/) 开始。

父元素 <module>

子元素 无

<finder-datasources> 元素

<finder-datasources> 指向您为插件创建的 IPluginAdaptor 实现的 Java 类文件。

您将设置用户访问插件的方式，并为 Orchestrator 通过插件执行的各种查找程序调用设置超时。下面这些来自 IPluginFactory 接口的查找程序方法将分别应用不同的超时。

<finder-datasources> 元素具有以下属性。

属性	值	描述
name	字符串	标识 <finder> 元素的 datasource 属性的数据源。等同于 XML id。
adaptor-class	Java 类	指向您定义的用于创建插件适配器的 IPluginAdaptor 实现，例如 com.vmware.plugins.sample.Adaptor。
concurrent-call	true（默认）或 false	允许多个用户同时访问适配器。如果插件不支持并发调用，则必须将 concurrent-call 设置为 false。
anonymous-login-mode	never（默认）或 always	向插件传递用户的用户名和密码，或不传递用户名和密码。
timeout-fetch-relation	数字；默认值为 30 秒	适用于来自 findRelation() 的调用
timeout-find-all	数字；默认值为 60 秒	适用于来自 findAll() 的调用
timeout-find	数字；默认值为 60 秒	适用于来自 find() 的调用

属性	值	描述
timeout-has-children-in-relation	数字；默认值为 2 秒	适用于来自 findChildrenInRelation() 的调用
timeout-execute-plugin-command	数字；默认值为 30 秒	适用于来自 executePluginCommand() 的调用

父元素 <module>

子元素 无子元素

<inventory> 元素

<inventory> 元素为显示在 Orchestrator 客户端 **Inventory** 视图和对象选取对话框中的插件定义层次结构列表的根对象。

<inventory> 元素不表示插入应用程序中的对象，而是将插件本身作为 Orchestrator 脚本 API 中的对象来表示。<inventory> 元素具有以下属性。

属性	值	描述
type	Orchestrator 对象类型	表示对象层次结构根对象的 <finder> 元素的类型

父元素 <module>

子元素 无

<finders> 元素

<finders> 元素是所有 <finder> 元素的容器。

<finders> 元素没有属性。

父元素 <module>

子元素 <finder>

<finder> 元素

<finder> 元素会在 Orchestrator 客户端中展示通过插件找到的一类对象。

<finder> 元素可标识用于定义对象查找程序展示的对象的 Java 类。<finder> 元素定义对象出现在 Orchestrator 客户端界面的方式。此外，它还可标识 Orchestrator 脚本 API 定义的、用以展示此对象的脚本对象。

查找程序作为由不同类型的插件技术使用的对象格式之间的接口使用。

属性	值	描述
type	Orchestrator 对象类型	由查找程序展示的对象类型
datasource	<finder-datasource name> 属性	通过使用数据源 refid 标识用于定义对象的 Java 类
dynamic-finder	Java 方法	定义在 IDynamicFinder 实例中实现的自定义查找程序方法，以通过编程方式返回查找程序的 ID 和属性，而无需在 vso.xml 文件中对其进行定义
hidden	true 或 false（默认）	如果为 true，则在 Orchestrator 客户端中隐藏查找程序
image	图形文件的路径	用于在 Orchestrator 客户端的层次结构列表中表示查找程序的 16x16 图标

属性	值	描述
java-class	Java 类的名称	用于定义查找程序找到并映射到脚本对象的对象的 Java 类
script-object	<scripting-object type> 属性	要将此查找程序映射到其中的 <scripting-object> 类型（如果有）

父元素 <finders>

子元素

- <properties>
- <relations>
- <id>
- <inventory-children>

<properties> 元素

<properties> 元素是 <finder> <property> 元素的容器。

<properties> 元素没有属性。

父元素 <finder>

子元素 <property>

<property> 元素

<property> 元素可通过使用 OGNL 表达式将已找到对象的属性映射到 Java 属性或方法调用。

实现插件工厂时，可以调用 SDKFinderProperty 类的方法以获取属性供插件工厂实现进行处理。

可以在 Orchestrator 客户端的视图中显示或隐藏对象属性。此外，还可以使用枚举定义对象属性。

<property> 元素具有以下属性。

属性	值	描述
name	查找程序名称	用于存储元素的 FinderResult 的名称。
display-name	查找程序名称	显示的属性名称
bean-property	属性名称	使用 bean-property 属性可标识通过 get 和 set 操作获取的属性。如果将某个属性的名称标识为 MyProperty，则插件将定义 getMyProperty 和 setMyProperty 操作。 您可以设置 bean-property 或 property-accessor 中的任何一个，但不能同时设置两个。
property-accessor	用于从对象获取属性值的方法	property-accessor 属性可用于定义 OGNL 表达式以验证对象的属性。 您可以设置 bean-property 或 property-accessor 中的任何一个，但不能同时设置两个。
show-in-column	true（默认）或 false	如果为 true，此属性将显示在 Orchestrator 客户端结果表中
show-in-description	true（默认）或 false	如果为 true，此属性将显示在对象描述中

属性	值	描述
hidden	true 或 false（默认）	如果为 true，则在所有情况下，此属性均被隐藏
linked-enumeration	枚举名称	将某个查找程序属性与枚举关联

父元素 <properties>

子元素 无

<relations> 元素

<relations> 元素是 <finder> <relation> 元素的容器。

<relations> 元素没有属性。

父元素 <finder>

子元素 <relation>

<relation> 元素

<relation> 元素可定义对象与其他对象相关的方式。

关系名称在 <inventory-children> 元素中定义。

<relation> 元素具有以下属性。

属性	值	描述
name	关系名称	此关系的名称
type	Orchestrator 对象类型	按此关系与另一个对象相关的对象的类型

父元素 <relations>

子元素 无

<id> 元素

<id> 元素可定义一种方法，用于获取查找程序标识的对象的 ID。

<id> 元素具有以下属性。

属性	值	描述
accessor	方法名称	accessor 属性可用于定义 OGNL 表达式以验证对象的 ID。

父元素 <finder>

<inventory-children> 元素

<inventory-children> 元素定义列表的层次结构，该列表显示 Orchestrator 客户端 **Inventory** 视图和对象选取框中的对象。

<inventory-children> 元素没有属性。

父元素 <finder>

子元素 <relation-link>

<relation-link> 元素

<relation-link> 元素可定义父对象和子对象之间的层次结构。

<relation-link> 元素具有以下属性。

类型	值	描述
name	关系名称	关系名称的 refid

父元素 <inventory-children>

子元素 无

<events> 元素

<events> 元素是 <trigger> 和 <gauge> 元素的容器。

<events> 元素没有属性。

父元素 <module>

子元素

- <description>
- <trigger-properties>

<trigger> 元素

<trigger> 元素用于声明可对此查找程序使用的触发器。必须实现 IPluginAdaptor 的 registerEventPublisher() 和 unregisterEventPublisher() 方法才能设置触发器。

<trigger> 元素具有以下属性。

类型	值	描述
name	关系名称	此关系的名称

父元素 <events>

子元素 无

<trigger-properties> 元素

<trigger-properties> 元素是 <trigger-property> 元素的容器。

<trigger-properties> 元素没有属性。

父元素 <trigger>

子元素 <trigger-property>

<trigger-property> 元素

<trigger-property> 元素可定义用于标识触发器对象的属性。

<trigger-property> 元素具有以下属性。

类型	值	描述
name	触发器名称	触发器的名称
display-name	触发器名称	显示在 Orchestrator 客户端中的名称
type	触发器类型	用于定义触发器的对象类型

父元素 <trigger-properties>

子元素 无

<gauge> 元素

<gauge> 元素定义可用于此查找程序的量表。必须实现 IPluginAdaptor 的 registerEventPublisher() 和 unregisterEventPublisher() 方法才能设置量表。

<gauge> 元素具有以下属性。

类型	值	描述
name	触发器名称	触发器的名称
display-name	触发器名称	显示在 Orchestrator 客户端中的名称
type	触发器类型	用于定义触发器的对象类型

父元素 <events>

子元素 <gauge-properties>

<gauge-properties> 元素

<gauge-properties> 元素是 <gauge-property> 元素的容器。

<gauge-properties> 元素没有属性。

父元素 <gauge>

子元素 <gauge-property>

<gauge-property> 元素

<gauge-property> 元素可定义用于标识量表对象的属性。

<gauge-property> 元素具有以下属性。

类型	值	描述
name	量表名称	量表的名称
display-name	量表名称	显示在 Orchestrator 客户端中的名称
type	量表类型	用于定义量表的对象类型
min-value	数字	量表监视的最小阈值
unit	测量类型	测量单位
max-value	数字	量表监视的最大阈值

父元素 <gauge-properties>

子元素 无

<scripting-objects> 元素

<scripting-objects> 元素是 <object> 元素的容器。

<scripting-objects> 元素没有属性。

父元素 <module>

子元素 <object>

<object> 元素

<object> 元素可将插件技术的构造函数、属性和方法映射到 Orchestrator 脚本 API 公开的 JavaScript 对象类型。

有关对象命名约定的信息，请参见第 123 页，“命名插件对象”。

<object> 元素具有以下属性。

类型	值	描述
script-name	JavaScript 名称	类的脚本名称。必须全局唯一
java-class	Java 类	由此 JavaScript 类包装的 Java 类
create	true（默认）或 false	如果为 true，则可以新建一个此类的实例
strict	true 或 false（默认）	如果为 true，则只能调用在 vso.xml 文件中注释或声明的方法
description	文本	对象的描述
url	数字	一个指向详细信息的链接，例如指向官方 API 文档的链接

父元素 <scripting-objects>

子元素

- <constructors>
- <attributes>
- <methods>
- <singleton>

<constructors> 元素

<constructors> 元素是 <object> <constructor> 元素的容器。

<constructors> 元素没有属性。

父元素 <constructors>

子元素 <constructor>

<constructor> 元素

<constructor> 元素可定义构造函数方法。用于 API 文档。

<constructor> 元素没有属性。

父元素 <constructors>

子元素

- <description>
- <parameters>

构造函数 <parameters> 元素

<parameters> 元素是 <constructor> <parameter> 元素的容器。

<parameters> 元素没有属性。

父元素	<constructor>
子元素	<parameter>

构造函数 <parameter> 元素

<parameter> 元素可定义构造函数的参数。

<parameter> 元素具有以下属性。

类型	值	描述
name	字符串	在 API 文档中使用的参数名称
type	Orchestrator 参数类型	在 API 文档中使用的参数类型

父元素	<parameters>
子元素	无

<attributes> 元素

<attributes> 元素是 <object> <attribute> 元素的容器。

<attributes> 元素没有属性。

父元素	<object>
子元素	<attribute>

<attribute> 元素

<attribute> 元素可将 Java 类的属性从插件技术映射到 Orchestrator JavaScript 引擎公开的 JavaScript 属性。

<attribute> 元素具有以下属性。

类型	值	描述
java-name	Java 属性	Java 属性的名称
script-name	JavaScript 对象	相应 JavaScript 对象的名称
return-type	Java 对象类型	此属性返回的类型
read-only	true 或 false	如果为 true，则无法修改此属性
is-optional	true 或 false	如果为 true，则此字段可以为 Null
show-in-api	true 或 false	如果为 false，此属性将不会出现在 API 文档中

父元素	<attributes>
子元素	无

<methods> 元素

<methods> 元素是 <object> <method> 元素的容器。

<methods> 元素没有属性。

父元素 <object>

子元素 <method>

<method> 元素

<method> 元素可将插件技术中的 Java 方法映射到 Orchestrator JavaScript 引擎公开的 JavaScript 方法。

<method> 元素具有以下属性。

类型	值	描述
java-name	Java 方法	Java 方法签名的名称，其参数类型由圆括号括起，例如，getVms(DataStore)
script-name	JavaScript 方法	相应 JavaScript 方法的名称
return-type	Java 对象类型	此方法获取的类型
show-in-api	true 或 false	如果为 false，此方法将不会出现在 API 文档中

父元素 <methods>

子元素 <parameters>

方法 <parameters> 元素

<parameters> 元素是 <method> <parameter> 元素的容器。

<parameters> 元素没有属性。

父元素 <method>

子元素 <parameter>

方法 <parameter> 元素

<parameter> 元素可定义方法的输入参数。

<parameter> 元素具有以下属性。

类型	值	描述
name	字符串	参数名称
type	Orchestrator 参数类型	参数类型
is-optional	true 或 false	如果为 true，则值可以为 Null。

父元素 <parameters>

子元素 <description>

<singleton> 元素

<singleton> 元素可用于创建单个 <script-object> 实例。不能对来自 JavaScript 脚本的单个对象进行实例化。

<singleton> 元素具有以下属性。

类型	值	描述
script-name	JavaScript 对象	相应 JavaScript 对象的名称
datasource	Java 对象	此 JavaScript 对象的源 Java 对象

父元素 <object>

子元素 无

<enumerations> 元素

<enumerations> 元素是 <enumeration> 元素的容器。

<enumerations> 元素没有属性。

父元素 <module>

子元素 <enumeration>

<enumeration> 元素

<enumeration> 元素可定义适用于某个类型的所有对象的通用值。

如果某种类型的所有对象需要某个特定属性，并且该属性的值具有限定的范围，则可以将不同的值定义为枚举条目。例如，如果某一类对象需要 color 属性，并且如果只能使用红色、蓝色和绿色，则可以定义三个枚举条目来定义这三个颜色值。这些条目将定义为枚举元素的子元素。

<enumeration> 元素具有以下属性。

类型	值	描述
type	Orchestrator 对象类型	枚举类型

父元素 <enumerations>

子元素

- <url>
- <description>
- <entries>

<url> 元素

<url> 元素提供指向有关枚举的文档的 URL。

<url> 元素没有属性。在 <url> 和 </url> 标记之间提供 URL。

父元素 <enumeration>

子元素 无

<entries> 元素

<entries> 元素是 <enumeration> <entry> 元素的容器。

<entries> 元素没有属性。

父元素 <enumeration>

子元素 <entry>

<entry> 元素

<entry> 元素可为枚举属性提供值。

<entry> 元素具有以下属性。

类型	值	描述
id	文本	对象用于将枚举条目设置为属性的标识符
name	文本	条目名称

父元素 <enumeration>

子元素 无

命名插件对象

必须为插件在插件技术中找到的每个对象提供一个唯一标识符。可以在 vso.xml 文件和脚本对象的 <finder> 元素中定义对象名称。

在工厂实现中定义的插件查找程序操作可用于查找插件技术中的对象。当插件找到对象时，您可以在 Orchestrator 工作流中使用它们，而且还可将它们从一个工作流元素传递到另一个元素。要允许对象在工作流中的各元素之间传递，必须为每个对象提供一个唯一标识符。

Orchestrator 服务器仅存储所处理的每个对象的类型和标识符，而不会存储有关 Orchestrator 获取对象的位置或方法的相关信息。您必须在插件实现中采用一致的方式命名对象，以便您可以跟踪从插件获取的对象。

如果工作流正在运行时 Orchestrator 服务器关闭，则重新启动服务器之后，工作流将恢复到其在服务器停止时所运行的工作流元素的位置。通过使用对象的标识符，工作流可检索服务器停止时该元素正在处理的任何对象。这就是必须为通过插件找到的所有对象提供一个唯一标识符的原因。

插件对象命名约定

在对插件中的所有对象进行命名时，必须遵循 Java 类命名约定。

- 对名称中的每个单词采用首字母大写
- 不要使用空格分隔单词
- 对于字母，仅使用标准字符 A 到 Z 和 a 到 z
- 不要使用特殊字符，如重音符号
- 不要将数字作为名称的第一个字符
- 字符数尽可能少于 10 个

表 6-1 显示适用于不同对象类型的其他规则。

表 6–1 插件对象命名规则

对象类型	命名规则
插件	■ 在 vso.xml 文件的 <module> 元素中定义。 ■ 必须遵循 Java 类命名约定。 ■ 插件名称必须唯一。不能在 Orchestrator 服务器中运行两个名称相同的插件。
查找程序对象	■ 在 vso.xml 文件的 <finder> 元素中定义。 ■ 必须遵循 Java 类命名约定。 ■ 查找程序名称在插件中必须唯一
脚本对象	■ 在 vso.xml 文件的 <scripting-object> 元素中定义。 ■ 必须遵循 Java 类命名约定。 ■ 脚本对象名称在 Orchestrator 服务器中必须唯一 ■ 为避免脚本对象名称与其他插件的脚本对象名称发生冲突，请始终使用插件的名称作为脚本对象名称的前缀

重要事项 鉴于工作流引擎处理数据序列化的方式，请不要在对象名称中使用以下字符串序列。

- `#;#`
- `#,#`
- `#=#`

在对象标识符中使用这些字符序列会造成工作流引擎错误分析工作流，并会在工作流运行时导致意外行为。

构造对象名称

Orchestrator 会根据您在 vso.xml 文件中定义的插件和查找程序名称构造唯一的对象名称。

例如，如果您定义一个名为 SolarSystem 的插件和一个名为 Planet 的查找程序（用于查找 Planet 类型的对象），则 Planet 查找程序对象的唯一名称即为 SolarSystem:Planet。

将 Planet 对象的 Java 类映射到 JavaScript 的脚本对象的唯一名称为 SolarSystemPlanet。

*.dar 文件的结构

您通过标准 Java 归档 (JAR) 或 ZIP 文件形式交付已完成的插件，但必须将其重命名为 *.dar。然后，将此 *.dar 文件导入到 Orchestrator 服务器中。

*.dar 归档文件的内容必须遵循以下文件夹结构和命名约定：

表 6–2 vso.xml 文件的结构

文件夹和文件	描述
/VSO-INF/vso.xml	定义插件技术中的对象到 Orchestrator 对象的映射。
/lib/	包含插件技术以及插件适配器和工厂接口实现的相关 JAR 文件。
/resources/	包含插件适配器和工厂实现所需的资源文件，如映像、脚本、HTML 段，等等。可以将资源组织到子文件夹。例如，/resources/images/ 或 /resources/scripts/。

要将插件导入到 Orchestrator 服务器中，请使用 Orchestrator 配置界面。有关如何使用配置界面导入插件的说明，请参见《VMware vCenter Orchestrator 管理指南》。

创建 Orchestrator 插件

要使用 Orchestrator 创建管理外部应用程序的插件，必须创建插件适配器和插件工厂，并在 `vso.xml` 文件中将插入应用程序中的对象映射到 Orchestrator 对象。

创建插件的步骤包含多个子步骤。下述步骤通过分析 Java 应用程序示例的 Java 类和 `vso.xml` 文件来演示插件创建过程。下述步骤检查的应用程序代表太阳系，其中包含代表太阳、行星以及各自卫星的 Java 对象，同时还定义了可对这些对象执行的不同操作。通过此应用程序的 Orchestrator 插件，您可以使用 Orchestrator 管理 Solar System 应用程序。通过该插件，您可以在 **Inventory** 视图中查看对象、通过 Web 视图访问这些对象、对这些对象执行操作以及运行自动执行 Solar System 操作的工作流。

步骤

1 获取要插入到 Orchestrator 中的 Java 应用程序第 125 页，

要创建插件，必须具有一个可供 Orchestrator 管理的 Java 应用程序。

2 检查示例插件适配器第 127 页，

要创建插件适配器，需要通过 Orchestrator 插件 API 创建实现 `IPluginAdaptor` 接口的 Java 类。

3 检查示例插件工厂第 129 页，

要创建插件工厂，需要通过 Orchestrator 插件 API 创建实现 `IPluginFactory` 接口的 Java 类。

4 在 vso.xml 文件中映射应用程序第 133 页，

`vso.xml` 文件定义 Orchestrator 访问以及与插件技术交互的方式。它可将插件技术中的对象和操作映射到 Orchestrator 对象和操作。

5 创建插件 *.dar 归档文件第 138 页，

创建插件的最后一个阶段是创建可导入 Orchestrator 中的 *.dar 文件。

获取要插入到 Orchestrator 中的 Java 应用程序

要创建插件，必须具有一个可供 Orchestrator 管理的 Java 应用程序。

前提条件

下列步骤使用 Solar System 示例应用程序来演示如何创建插件。可从 Orchestrator 文档主页下载 Orchestrator 示例的 ZIP 文件，该文件包含插件的 `*.dar` 文件、Solar System 示例应用程序的源文件以及插件实现的源文件。有关下载 Orchestrator 示例包的位置的详细信息，请参见第 5 页，“示例应用程序”。

步骤

1 从 Orchestrator 文档主页下载 Orchestrator 示例的 ZIP 文件。

2 将压缩包解压到适当的位置。

3 导航到以下位置以查看 Solar System 插件文件。

`/install-directory/vcenter_orchestrator_examples/Plug-Ins`

`/Plug-Ins` 文件夹包含以下文件和文件夹。

- `/vmware-vmosdk-solarsystem.dar`，是包含已完成插件的 `*.dar` 归档文件。
- `/SolarSystem`，包含要检查的 Solar System 应用程序的源文件。
- `/VSOSDK-SolarSystem`，包含 Orchestrator 插件 API 接口的实现，用于定义 Solar System 应用程序的插件适配器和工厂。

4 （可选）解压 `vmware-vmosdk-solarsystem.dar` 归档文件以查看已完成插件的内容。

Solar System 示例应用程序的组件

此 Solar System 示例应用程序复制太阳系的结构，包括代表恒星、行星、卫星的对象以及可以对这些对象执行的操作。

/SolarSystem/src/com/vmware/solarsystem 文件夹包含以下文件。

表 6–3 Solar System 插件示例应用程序的源文件

类名称	描述
CelestialBody.java	定义一般天体的可序列化类，利用 get 和 set 方法获取和设置对象的 name 和 id 的值。
Moon.java	用于扩展 CelestialBody 及定义各种方法，用于获取和设置其体积以及获取和设置其绕之运转的行星的标识符。
Planet.java	用于扩展 CelestialBody 及定义各种方法，用于获取和设置其周长和重力级别，获取、设置、添加和删除卫星以及获取和设置其绕之运转的恒星的标识符。
Star.java	用于扩展 CelestialBody 及定义各种方法，用于获取和设置其周长和表面温度，获取、设置、添加和删除在其轨道中运转的行星。
SolarSystemRepository.java	用于创建一个 Solar System 存储库实例，其中包含 Star、Planet 和 Moon 类的实例。这些实例代表太阳和行星以及其各自在地球太阳系中运转的卫星。

/VSOSDK-SolarSystem/src/com/vmware/orchestrator/api/sample/solarsystem 文件夹包含以下文件。

表 6–4 Solar System 插件实现的源文件

类名称	描述
SolarSystemAdapter.java	用于实现 IPluginAdaptor 接口，该接口定义 Solar System 应用程序进入 Orchestrator 的入口点。
SolarSystemEventGenerator.java	一种 Java 类，可创建 IPluginEventPublisher 实例并定义将事件发布到 Orchestrator 策略引擎的各种方法。
SolarSystemFactory.java	一种 IPluginFactory 接口实现，用于定义 Orchestrator 如何通过插件查找对象，以及如何对这些对象执行操作。

注意 Solar System 示例应用程序的源文件仅供参考。如果将 vmware-vmosdk-solarsystem.dar 文件导入到 Orchestrator 中，则无需构建 Solar System Java 代码或使用源文件来创建此插件。

安装 Solar System 插件 JAR 文件

要运行 Solar System 示例插件，必须下载并在插件中安装第三方 Java 归档文件。

可免费下载 Solar System 插件所需的 JAR 文件。

步骤

1 从 http://apache.mirror.testserver.li/xml/commons/xml-commons-external-1.3.04-bin.zip 下载 XML Commons API Java 归档文件，即 xml-apis.jar。

   xml-apis.jar 归档文件包含在分发版中。

2 解压 Solar System dar 归档文件。

3 将 `xml-apis.jar` 归档文件复制到以下 Solar System `dar` 归档位置。

<solar_system_安装_目录>`/lib/`

4 重新将 Solar System `dar` 归档文件打包。

您已安装 Solar System 示例插件所需的 Java 归档文件。

下一步

可以检查 Solar System 示例插件的类。

检查示例插件适配器

要创建插件适配器，需要通过 Orchestrator 插件 API 创建实现 `IPluginAdaptor` 接口的 Java 类。

下列步骤介绍了创建插件适配器所涉及的步骤。为了演示此过程，本部分列出了 Solar System 示例应用程序的 `SolarSystemAdapter` 类中的代码。可以从 Orchestrator 文档主页下载 Orchestrator 示例 ZIP 文件，以获取 Solar System 示例应用程序的源文件。

前提条件

下列步骤概述了开发插件适配器类的过程中的众多步骤。要了解有关插件适配器以及其他插件组件的作用的说明，必须阅读第 107 页，“插件的组件和架构”。

步骤

1 为插件适配器实现创建并保存一个名为 *<您的应用程序名称>*`Adaptor.java` 的 Java 文件。

在 Solar System 示例中，适配器类称为 `SolarSystemAdapter.java`。

2 声明一个软件包，使其包含插件适配器和工厂实现。

Solar System 示例声明以下软件包以包含适配器和工厂实现：

```
package com.vmware.orchestrator.api.sample.solarsystem;
```

3 使用 Java `import` 语句导入 Orchestrator 插件 API 接口、类和枚举。

```
import ch.dunes.vso.sdk.api.*;
```

4 导入适配器实现所需的任何其他类。

在 Solar System 示例中，适配器实现需要以下类：

```
import javax.security.auth.login.LoginException;
```

5 声明一个公共构造函数，用于通过 Orchestrator 插件 API 实现 `IPluginAdaptor` 接口。

Solar System 示例适配器声明了 `SolarSystemAdapter` 构造函数。

```
public class SolarSystemAdapter implements IPluginAdaptor {
}
```

6 创建一个实现 IPluginFactory 接口的插件工厂类实例。

通过调用 createPluginFactory() 方法，创建工厂实例。

Solar System 示例适配器调用 createPluginFactory() 方法来创建名为 factory 的 SolarSystemFactory 类的实例。

```
public class SolarSystemAdapter implements IPluginAdaptor {

private SolarSystemFactory factory;

public IPluginFactory createPluginFactory(String sessionID, String username,
String password, IPluginNotificationHandler notificationHandler)
throws SecurityException, LoginException, PluginLicenseException {

if (factory == null) {
factory = new SolarSystemFactory();
    }
return factory;
}
```

7 定义插件适配器管理的事件。

可以定义要在适配器实现中直接管理的事件。但是，Solar System 插件实现在名为 SolarSystemEventGenerator 的单独类中定义事件。

SolarSystemEventGenerator 类可定义以下三种方法：

- addPolicyElement()，用于向 Orchestrator 策略引擎注册事件。
- removePolicyElement()，用于删除 Orchestrator 策略引擎中的事件。
- generatareFlareEvent()，用于生成太阳耀斑事件，Orchestrator 使用量表监控该事件的等级。

SolarSystemAdapter 定义以下 getEventGenerator() 方法，用于获取 SolarSystemEventGenerator 的实例以提供对这些方法及其管理的事件的访问。

```
private SolarSystemEventGenerator getEventGenerator() { return
SolarSystemEventGenerator._solarSystemEventGenerator; }
```

8 如果 Orchestrator 监控插入应用程序中发生的事件，必须通过调用 IPluginAdaptor 接口的 registerEventPublisher() 方法注册 IPluginEventPublisher 接口的实例。

Solar System 示例适配器通过调用 SolarSystemEventGenerator 类的 addPolicyElement() 方法，创建以下 IPluginEventPublisher 实例，并将其添加到 Orchestrator 策略中。

```
public void registerEventPublisher(String type, String id,
IPluginEventPublisher publisher) {
getEventGenerator().addPolicyElement(type, id, publisher); }
```

9 通过调用 installLicenses() 方法以实例化 PluginLicense 对象的数组来安装 Orchestrator 访问插件技术所需的所有许可证。

```
public void installLicenses(PluginLicense[] licenses) throws
PluginLicenseException {
}
```

10 将插件名称传递到插件。

如有必要，`IPluginAdaptor` 接口的 `setPluginName()` 方法会从 `vso.xml` 文件获取该名称。

```
public void setPluginName(String pluginName) {
}
```

11 通过调用从 Orchestrator 策略引擎取消注册 `IPluginEventPublisher`。

Solar System 示例适配器通过调用 `unregisterEventPublisher()` 方法，取消对在步骤 8 中创建的 `IPluginEventPublisher` 实例的注册，并从 Orchestrator 策略引擎中将其移除。

```
public void unregisterEventPublisher(String type, String id,
IPluginEventPublisher publisher) {
getEventGenerator().removePolicyElement(type, id, publisher); }
```

您拥有一个插件适配器实现，可用于创建插件工厂、获取许可证、将插件名称传递到 Orchestrator，以及定义如何管理插件技术中发生的事件。

下一步

创建插件工厂实现以通过插件查找对象，并定义如何对这些对象执行操作。

检查示例插件工厂

要创建插件工厂，需要通过 Orchestrator 插件 API 创建实现 `IPluginFactory` 接口的 Java 类。

前提条件

下列步骤概述了开发插件工厂类的过程中的众多步骤。要了解有关插件工厂以及其他插件组件的作用的说明，必须阅读第 107 页，“插件的组件和架构”。

步骤

1 为插件工厂实现创建并保存一个名为 *<您的应用程序名称>*`Factory.java` 的 Java 文件。

在 Solar System 示例中，工厂类称为 `SolarSystemFactory.java`。

2 声明一个软件包，使其包含插件适配器和工厂实现。

Solar System 示例声明以下软件包以包含适配器和工厂实现：

```
package com.vmware.orchestrator.api.sample.solarsystem;
```

3 使用 Java `import` 语句将 Orchestrator 插件 API 类导入到工厂实现中。

```
import ch.dunes.vso.sdk.api.*;
```

4 使用 Java `import` 语句导入 Solar System 应用程序。

```
import com.vmware.solarsystem.*;
```

5 导入适配器实现所需的任何其他类。

在 Solar System 示例中，工厂实现需要以下类：

```
import java.util.*;
import org.apache.log4j.Logger;
```

6 声明一个公共构造函数，用于通过 Orchestrator 插件 API 实现 `IPluginFactory` 接口。

Solar System 示例适配器声明了 `SolarSystemFactory` 构造函数。

```
public class SolarSystemFactory implements IPluginFactory {
}
```

7 设置日志记录，以便 Orchestrator 可以跟踪插件技术中发生的事件。

```
private Logger log = Logger.getLogger(this.getClass());
```

8 通过调用 `IPluginFactory` 接口的 `executePluginCommand()` 方法及定义命令行为，定义 Orchestrator 如何对通过插件找到的对象执行操作。

`SolarSystemFactory` 示例实现不支持任何命令，因此无法实现 `executePluginCommand()` 方法。

```
public void executePluginCommand(String cmd) throws PluginExecutionException {
}
```

9 通过实现 `IPluginFactory` 接口的 `find()` 方法，定义 Orchestrator 如何通过插件按照对象的名称和类型查找对象。

`SolarSystemFactory` 示例实现调用由 `SolarSystemRepository` 类定义的方法以按照各个对象的类型和标识符获取 `Star`、`Planet` 和 `Moon` 对象。

```
public Object find(String type, String id) {
log.debug("find:" + type + ", " + id);
if (type.equals("Star")) {
return SolarSystemRepository.getUniqueInstance().getStar();
  }
else if (type.equals("Planet")) {
return SolarSystemRepository.getUniqueInstance().getPlanetById(id);
  }
else if (type.equals("Moon")) {
return SolarSystemRepository.getUniqueInstance().getMoonById(id);
  }
else if (type.equals("Galaxy")) {
return null; // No object for galaxy defined yet
  }
else {
throw new IndexOutOfBoundsException("Type " + type + "
+ unknown for plugin SolarSystem");
  }
}
```

10 通过实现 IPluginFactory 接口的 findAll() 方法，定义查询规则以按照您自己的查询条件通过插件查找对象。

SolarSystemFactory 示例实现将返回一个包含所有给定类型对象的列表。例如，它可能会返回一个包含所有 Planet 类型对象的列表。

```
public QueryResult findAll(String type, String query) {
log.debug("findAll:" + type + ", " + query);
List list; // The list can contain any element from the plug-in
if (type.equals("Star")) {
list = new Vector();
list.add(SolarSystemRepository.getUniqueInstance().getStar());
  }
else if (type.equals("Planet")) {
list = SolarSystemRepository.getUniqueInstance().getAllPlanets();
  }
else if (type.equals("Moon")) {
list = SolarSystemRepository.getUniqueInstance().getAllMoons();
  }
else if (type.equals("Galaxy")) {
list = new Vector();
  }
else {
throw new IndexOutOfBoundsException("Type " + type +
" unknown for SolarSystem plug-in ");
  }
return new QueryResult(list);
}
```

findAll() 方法的 SolarSystemFactory 类的实现不会定义自定义查询，因此它将在 QueryResult 对象中返回一个包含所有给定类型对象的列表。

11 通过实现 IPluginFactory 接口的 findRelation() 方法，定义 Orchestrator 如何通过插件按其要查找的对象与其他对象的关系查找这些对象。

SolarSystemFactory 示例实现 findRelation() 以返回绕给定恒星运转的所有行星，或绕给定行星运转的所有卫星。

```
public List findRelation(String parentType, String parentId, String relationName) {
log.debug("findRelation:" + parentType + ", " + parentId + ", " + relationName);
if (parentId == null) {
List<Star> list = new Vector<Star>();
list.add(SolarSystemRepository.getUniqueInstance().getStar());
return list;
  }
if (parentType.equals("Star")) {
if (relationName.equals("OrbitingPlanets")) {
return SolarSystemRepository.getUniqueInstance().getAllPlanets();
  }
else {
throw new IndexOutOfBoundsException("Unknown relation name: "
+ relationName);
    }
  }
if (parentType.equals("Planet")) {
if (relationName.equals("OrbitingMoons")) {
Planet parentPlanet =
SolarSystemRepository.getUniqueInstance().getPlanetById(parentId);
if (parentPlanet != null) {
return parentPlanet.getMoons();
        }
return null;
  }
else {
throw new IndexOutOfBoundsException("Unknown relation name: "
+ relationName);
    }
  }
else {
return null;
  }
}
```

SolarSystemFactory 示例的 findRelation() 方法将返回按 OrbitingPlanets 和 OrbitingMoons 关系类型与父对象相关的对象的列表，这些关系类型在 vso.xml 文件中定义。

12 通过实现 IPluginFactory 接口的方法，发现对象是否具有某个关系类型的子对象。

SolarSystemFactory 示例将实现 hasChildrenInRelation() 以确定对象是否具有按给定类型与其相关的子对象。

```
public HasChildrenResult hasChildrenInRelation(String parentType,
String parentId, String relationName) {
return HasChildrenResult.Unknown;
}
```

可能的返回值包括 Yes、No 和 Unknown。

您已拥有一个插件工厂实现，可用于定义 Orchestrator 如何通过插件查找对象以及如何对这些对象执行操作。

下一步

在 vso.xml 文件中将应用程序对象映射到 Orchestrator 对象。

在 vso.xml 文件中映射应用程序

vso.xml 文件定义 Orchestrator 访问以及与插件技术交互的方式。它可将插件技术中的对象和操作映射到 Orchestrator 对象和操作。

前提条件

下列步骤将分析 Solar System 示例应用程序的 vso.xml 文件的主要部分。要了解有关 vso.xml 文件的所有元素及其所有属性的完整说明，请参见第 111 页，“vso.xml 插件定义文件的元素”。

步骤

1 设置插件第 133 页，

要创建插件，必须将 Orchestrator 指向相关 XML 架构定义、应用程序和插件的源文件，定义插件在 Orchestrator 启动时的行为，并为插件公开的对象的层次结构提供根对象。

2 定义用于查找对象的 Finder 元素第 134 页，

要使 Orchestrator 能够访问插入应用程序中的对象，必须定义插件查找这些对象的方式和位置。

3 定义枚举第 136 页，

可以通过定义枚举来设置应用于各种不同类型的所有对象的值。

4 将事件映射到 Orchestrator API 方法第 136 页，

为了使 Orchestrator 能够监控插入应用程序中的对象并对这些对象执行操作，需要将应用程序定义的事件映射到您添加到 Orchestrator 脚本 API 的方法。

5 将对象映射到 JavaScript 对象第 137 页，

为了使 Orchestrator 能够调用插入应用程序中的对象及其方法，需要将该应用程序定义的对象和方法映射到您添加到 Orchestrator 脚本 API 的方法。

设置插件

要创建插件，必须将 Orchestrator 指向相关 XML 架构定义、应用程序和插件的源文件，定义插件在 Orchestrator 启动时的行为，并为插件公开的对象的层次结构提供根对象。

步骤

1 创建并保存一个名为 vso.xml 的文件。

2 设置 <module> 元素以提供有关插件的基本信息，包括指针和 Orchestrator 插件 XML 架构定义。

Solar System 示例 vso.xml 文件中的 <module> 元素将插件名称设置为 SolarSystem，还设置了版本号，并且在 *.dar 归档文件中提供了指向在 Orchestrator **Inventory** 视图和选取对话框中表示此插件的图标的路径。

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.vmware.com/support/orchestrator/plugin-4-0.xsd"
    name="SolarSystem" version="1.0.0" build-number="4" image="images/solarSystem-16x16.png">
```

3 在 <description> 元素中提供对插件的描述。

Solar System 示例描述如下。

```
<description>Example plug-in to a solar system application.</description>
```

4 在 <installation> 和 <action> 元素中设置当 Orchestrator 服务器启动时插件的行为。

只要检测到有新版本，Solar System 示例即会通过设置 version 模式启动插件，并在 *.dar 归档文件中提供指向 Orchestrator 组件软件包的路径。

```
<installation mode="version">
        <action type="install-package" resource="packages/com.vmware.solarsystem.package" />
    </installation>
```

5 在 <inventory> 元素中设置对象类型层次结构的根对象。

Solar System 插件将表示 Orchestrator 脚本 API 中的插件的层次结构的根对象定义为一个 Galaxy 类型的对象。现在，所有其他 Solar System 对象都与 Galaxy 对象相关。

```
<inventory type="Galaxy"/>
```

您已定义并设置用于标识插入到 Orchestrator 中的插件的元素。

下一步

定义 Orchestrator 如何通过插件使用 <finder> 元素查找对象。

定义用于查找对象的 Finder 元素

要使 Orchestrator 能够访问插入应用程序中的对象，必须定义插件查找这些对象的方式和位置。

前提条件

您必须已经创建 vso.xml 文件，并定义了 Orchestrator 标识插件的方式。

步骤

1 在 <finder-datasources> 元素中设置插件 <finder> 元素的数据源。

Solar System 插件 vso.xml 文件将数据源的名称设置为 solar-datasource，并将 <finder> 元素指向创建 SolarSystemFactory 实例的 SolarSystemAdapter 类。

```
<finder-datasources>
<finder-datasource name="solar-datasource"
adaptor-class=
                "com.vmware.orchestrator.api.sample.solarsystem.SolarSystemAdapter"
                anonymous-login-mode="internal"/>
</finder-datasources>
```

2 定义插件如何在 <finder> 元素中查找插件技术中的对象。

从 Solar System vso.xml 文件中提取的下列内容显示了 Star 类型的对象的 <finder> 元素。

```
<finders>
        <finder type="Star" datasource="solar-datasource"
                java-class="com.vmware.solarsystem.Star"
                script-object="Star" image="images/sun_16x16.png">
            [...]
        </finder>
            [...]
</finders>
```

Star 对象的 <finder> 元素从 <finder-datasource> 元素定义的数据源中获取这些对象的数据。Star 对象类型表示 com.vmware.solarsystem.Star 类的实例。

插件 <finder> 元素的主要特点是子元素可以映射对象的关系、属性和操作。

3 在 <id> 元素中获取或设置对象的标识符。

Solar System 示例通过调用 Solar System 应用程序的 CelestialBody 类定义的 getId() 方法来获取对象的标识符。

```
<id accessor="getId()" />
```

4 在 <relations> 元素中定义对象的关系。

Solar System 示例通过定义名为 OrbitingPlanets 的关系，使 Planet 类型的对象与此 <finder> 元素找到的 Star 对象相关。

```
<relations>
<relation type="Planet" name="OrbitingPlanets"/>
</relations>
```

5 根据对象与其父对象的关系在 **Inventory** 中设置这些对象的层次结构。

Solar System 示例将按照 OrbitingPlanets 关系将所有与 Star 对象类型相关的对象紧挨着放置在清单层次结构中该类型对象的下方。

```
<inventory-children>
<relation-link name="OrbitingPlanets"></relation-link>
</inventory-children>
```

6 在 <properties> 元素中设置对象的属性。

Solar System 示例为所有 Star 对象定义了 name、circumference 和 surfaceTemp 属性。通过 bean-property 属性，Orchestrator 可以在脚本 API 中创建 get 和 set 方法以获取和设置这些属性。在此示例中，脚本 API 将定义 getCircumference、setCircumference、getsurfaceTemp 和 setsurfaceTemp 方法。

```
<properties>
<property display-name="Name" name="name"
bean-property="name"/>
<property display-name="Circumference" name="circumference"
bean-property="circumference"/>
<property display-name="Surface Temperature" name="surfaceTemp"
bean-property="surfaceTemp"/>
</properties>
```

7 在 <events> 元素中设置对象的事件。

事件可能是量表或触发器。

在 Solar System 示例中，Star 对象用于定义生成太阳耀斑的方法。<gauge> 元素则监控 Star 对象生成的耀斑事件的值。

```
<events>
<gauge min-value="0" name="Flare" unit="number">
<description>Magnitude of the flare</description>
</gauge>
</events>
```

您已定义用于在插入应用程序中查找对象的 <finder> 元素。

下一步

定义枚举。

定义枚举

可以通过定义枚举来设置应用于各种不同类型的所有对象的值。

前提条件

必须已经设置了插件，并定义了 `<finder>` 元素。

步骤

1 在 `<enumerations>` 元素中定义对象类型的枚举。

Solar System 示例定义了多个枚举，以对 `Planet` 对象设置 `PlanetCategory` 枚举。

```
<enumerations>
<enumeration type="PlanetCategory">
<description>Define the category of a Planet</description>
  [...]
</enumeration>
```

2 定义枚举的条目，以便将值应用于给定对象类别中的不同对象。

Solar System 示例定义了表示不同类型行星的值。

```
<entries>
<entry id="gaz"
name="Huge Gaz">Huge planet with only gaz atmosphere.
No Physical core.</entry>
<entry id="earth"
name="Earth">You could live on this planet.</entry>
<entry id="desert"
name="Desert">Planet without water.</entry>
<entry id="ice"
name="Ice">Planet with water but completely frozen.</entry>
<entry id="other"
name="Other">Does not fit into any category.</entry>
</entries>
```

您已定义应用于某个特定类别中的所有对象的枚举。

下一步

将应用程序事件映射到 Orchestrator 脚本 API 中的方法。

将事件映射到 Orchestrator API 方法

为了使 Orchestrator 能够监控插入应用程序中的对象并对这些对象执行操作，需要将应用程序定义的事件映射到您添加到 Orchestrator 脚本 API 的方法。

在 `<scripting-objects><object>` 元素中标识要映射的事件。

前提条件

必须已经设置了插件，并定义了 `<finder>` 元素和枚举。

步骤

1 将定义应用程序事件的 Java 类映射到 JavaScript 对象。

SolarSystemEventGenerator 类为 SolarSystemAdapter 插件适配器实现定义 Solar System 应用程序事件。下列节选代码将这些事件映射到名为 _SolarSystemEventGenerator 的 JavaScript 对象。通过将 strict 属性设置为 true，Orchestrator 只能从 _SolarSystemEventGenerator JavaScript 对象调用方法。

```
<scripting-objects>
    <object script-name="_SolarSystemEventGenerator"
    java-class="com.vmware.orchestrator.api.sample.solarsystem.SolarSystemEventGenerator"
    strict="true">
    <description>The entry point to generate events</description>
    [...]
    </object>
    [...]
</scripting-objects>
```

2 （可选）如有必要，请将 JavaScript 对象表示为单个对象。

```
<singleton script-name="SolarSystemEventGenerator"
    datasource="solar-datasource"/>
```

3 在 <object><methods> 元素中将插入应用程序中的方法映射到可以在 Orchestrator API 中调用的 JavaScript 方法。

Solar System Star 对象定义生成太阳耀斑事件的方法。下列节选代码将此方法映射到具有相同名称的 JavaScript 方法，并在 JavaScript 方法中设置其参数。

```
<methods>
    <method script-name="generatareFlareEvent"
      java-name="generatareFlareEvent">
      <description>Start a Solar Flare</description>
    <parameters>
      <parameter name="star"
      type="Star">The star which generates the event</parameter>
    <parameter name="magnitude"
      type="number" >The magnitude of the flare</parameter>
      </parameters>
    </method>
</methods>
```

您已将应用程序事件映射到 Orchestrator 脚本 API 中的方法。

下一步

将应用程序的对象映射到 Orchestrator 脚本 API 中的 JavaScript 对象。

将对象映射到 JavaScript 对象

为了使 Orchestrator 能够调用插入应用程序中的对象及其方法，需要将该应用程序定义的对象和方法映射到您添加到 Orchestrator 脚本 API 的方法。

在 <scripting-objects><object> 元素中标识要映射的对象。

前提条件

您必须已设置插件，已定义 <finder> 元素和枚举，并已将应用程序的事件映射到 Orchestrator API 中的方法。

步骤

1 将用于定义对象的 Java 类映射到 `<object>` 元素中的 JavaScript 对象。

Solar System 示例中的 `Star` 对象定义了用于执行不同操作的方法。Solar System 示例 `vso.xml` 文件中的下列节选代码将应用程序中的 `Star` Java 类映射到同名的 JavaScript 对象。通过将 `create` 属性设置为 `false`，Orchestrator 便无法创建此对象的实例。通过将 `strict` 属性设置为 `true`，Orchestrator 便只能调用 `Star` JavaScript 对象中的方法。

```
<object script-name="Star" java-class="com.vmware.solarsystem.Star"
create="false" strict="true">
<description>A star, center of a solar system</description>
  [...]
</object>
```

2 在 `<object><attributes>` 元素中将对象的属性映射到 JavaScript 属性。

Solar System 示例 `vso.xml` 文件中的下列节选代码将 `Star` 对象的 Java 属性映射到同名的 JavaScript 属性。

```
<attributes>
<attribute script-name="id" java-name="id"
return-type="string">The unique Id of the star</attribute>
<attribute script-name="name" java-name="name"
       return-type="string">The name of the star</attribute>
     <attribute script-name="circumference" java-name="circumference"
       return-type="number">Circumference of the star</attribute>
     <attribute script-name="temperature" java-name="surfaceTemp"
       return-type="number">The temperature on the star's surface</attribute>
</attributes>
```

3 在 `<object><methods>` 元素中将对象的方法映射到 JavaScript 方法。

Solar System 示例 `vso.xml` 文件中的下列节选代码将 `Star` 对象的一种 Java 方法映射到同名的 JavaScript 方法。

```
<methods>
<method script-name="addPlanet" java-name="addPlanet">
<description>Add new planet to the solar system</description>
       <parameters>
<parameter type="Planet" name="planet">The planet to add</parameter>
</parameters>
</method>
</methods>
```

您已将应用程序的对象映射到 Orchestrator 脚本 API 中的 JavaScript 对象。

下一步

创建包含插件组件的 `*.dar` 归档文件。

创建插件 *.dar 归档文件

创建插件的最后一个阶段是创建可导入 Orchestrator 中的 *.dar 文件。

`*.dar` 归档文件是重命名为 `*.dar` 的标准 `*.jar` Java 归档文件。`*.dar` 归档文件必须符合标准的文件和文件夹结构。

前提条件

您已将应用程序插入到 Orchestrator 中，已经创建了适配器和工厂实现，并且已经在 `vso.xml` 文件中将应用程序映射到 Orchestrator 对象。

步骤

1 创建一个要在其中创建 `*.dar` 归档文件的工作目录。

2 在工作目录的根目录处创建一个名为 `VSO-INF` 的文件夹。

3 将 `vso.xml` 文件复制到 `VSO-INF` 中。

4 在工作目录的根目录处创建一个名为 `lib` 的文件夹。

5 将包含插入应用程序的类以及插件适配器和工厂实现的类的 JAR 文件复制到 `lib` 中。

6 在工作目录的根目录处创建一个名为 `resources` 的文件夹。

7 在 `resources` 文件夹中创建一个名为 `images` 的文件夹。

8 将图标复制到 `resources/images` 中。

  这些图标用于表示 Orchestrator **Inventory** 视图和选取对话框中的各种插入应用程序对象。

9 在 `resources` 文件夹中创建一个名为 `packages` 的文件夹。

10 将 Orchestrator 软件包复制到 `resources/packages` 中。

  这些软件包可包含工作流、操作、策略等内容，用于与插入应用程序交互。

11 在工作目录的根目录处创建一个名为 `web-content` 的文件夹。

12 将为应用程序创建的所有 Web 视图的组件复制到 `web-content` 文件夹中。

13 创建一个 Java 归档文件，用于包含上述所有文件夹和文件。

  例如，在命令行处运行以下 `jar` 命令。

```
jar -cf myDarFile VSO.INF lib resources web-content
```

14 将 Java 归档文件从 `*.jar` 重命名为 `*.dar`。

15 将插件导入到 Orchestrator 服务器中。

  可以使用两种方法导入插件：

  - 将 `*.dar` 归档文件复制到 Orchestrator 插件文件夹中，位置如下。

    安装-目录`\app-server\server\vmo\plugins`

  - 使用 Orchestrator 配置界面。有关如何使用配置界面导入插件的信息，请参见《VMware vCenter Orchestrator 管理指南》。

您已创建包含插件的 `*.dar` 归档文件，并已将该归档文件导入到 Orchestrator 中。

示例 6–1 Solar System *.dar 归档文件的内容

为了说明 `*.dar` 归档文件的内容和结构，Solar System `vmware-vmosdk-solarsystem.dar` 示例归档文件包含以下文件夹和文件。

- `/lib`，包含以下 JAR 归档文件：
  - `vmware-solarsystem.jar`，包含 Solar System 应用程序。
  - `vmware-vmosdk-solarsystem.jar`，包含 Solar System 应用程序的插件适配器和工厂实现的类。
- `/resources`
  - `/images`，包含表示 Orchestrator **Inventory** 视图中的各种 Solar System 应用程序对象的图标。
  - `/packages`，包含名为 `com.vmware.solarsystem.package` 的 Orchestrator 软件包。该软件包包含使 Orchestrator 与 Solar System 应用程序实现交互的工作流、策略、操作和 Web 视图。
- `/VSO-INF/vso.xml`，此 XML 文件可将 Solar System 应用程序映射到 Orchestrator 对象。

下一步

您可以在“Inventory”视图中访问插入应用程序的对象，以对这些对象执行操作。此外，还可以使用映射到 Orchestrator 脚本 API 的对象和方法来创建工作流、操作、策略、Web 视图等，以便通过插件与这些对象交互。如果已经创建了工作流、操作、策略和 Web 视图，则可以将其作为软件包添加到插件 `*.dar` 文件中。

Orchestrator 插件 API 参考

Orchestrator 插件 API 可在您开发 `IPluginAdaptor` 和 `IPluginFactory` 实现以创建插件时，定义要实现和扩展的 Java 接口和类。

IDynamicFinder 接口

`IDynamicFinder` 接口通过编程方式返回查找程序的 ID 和属性，而不是在 `vso.xml` 文件中定义 ID 和属性。

`IDynamicFinder` 接口可定义以下方法。

方法	返回	描述
`getIdAccessor(java.lang.String type)`	`java.lang.String`	提供一个 OGNL 表达式，以便通过编程方式获取对象 ID。
`getProperties(java.lang.String type)`	`java.util.List<SDKFinderProperty>`	通过编程方式提供一个对象属性列表。

IPluginAdaptor 接口

实现 `IPluginAdaptor` 接口可管理插件工厂、事件和监视程序。`IPluginAdaptor` 接口可在插件和 Orchestrator 服务器之间定义一个适配器。

`IPluginAdaptor` 实例负责会话管理。`IPluginAdaptor` 接口可定义以下方法。

方法	返回	描述
`addWatcher(PluginWatcher watcher)`	Void	添加一个监视程序以监控特定事件
`createPluginFactory(java.lang.String sessionID, java.lang.String username, java.lang.String password, IPluginNotificationHandler notificationHandler)`	`IPluginFactory`	创建 `IPluginFactory` 实例。Orchestrator 服务器使用该工厂，通过其 ID、与其他对象的关系等方式从插件技术获取对象。 通过会话 ID，可以标识正在运行的会话。例如，用户可以同时登录两个不同的 Orchestrator 客户端，并同时运行两个会话。 同样，启动工作流时会创建一个独立于启动工作流的客户端的会话。即使关闭 Orchestrator 客户端，工作流也会继续运行。
`installLicenses(PluginLicense[] licenses)`	Void	为 VMware 提供的标准插件安装许可证信息
`registerEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)`	Void	对清单中的元素设置触发器和量表
`removeWatcher(java.lang.String watcherId)`	Void	移除监视程序
`setPluginName(java.lang.String pluginName)`	Void	从 `vso.xml` 文件获取插件名称
`setPluginPublisher(IPluginPublisher pluginPublisher)`	Void	设置插件的发布者

方法	返回	描述
uninstallPluginFactory(IPluginFactory plugin)	Void	卸载插件工厂。
unregisterEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)	Void	移除清单中某个元素的触发器和量表

IPluginEventPublisher 接口

IPluginEventPublisher 接口在事件通知总线上为要监视的 Orchestrator 策略发布量表和触发器。

IPluginEventPublisher 接口可定义以下方法。

类型	值	描述
pushGauge(java.lang.String type, java.lang.String id, java.lang.String gaugeName, java.lang.String deviceName, java.lang.Double gaugeValue)	Void	在事件通知总线上发布量表
pushTrigger(java.lang.String type, java.lang.String id, java.lang.String triggerName, java.util.Properties additionalProperties)	Void	在事件通知总线上发布触发器

IPluginFactory 接口

IPluginAdaptor 返回 IPluginFactory 实例。IPluginFactory 实例运行插件应用程序中的命令，并通过插件查找要在其上执行 Orchestrator 操作的对象。

IPluginFactory 接口可定义以下方法。

方法	返回	描述
executePluginCommand(java.lang.String cmd)	Void	通过插件执行命令
find(java.lang.String type, java.lang.String id)	java.lang.Object	通过插件按其 ID 和类型查找对象
findAll(java.lang.String type, java.lang.String query)	QueryResult	通过插件查找与查询字符串匹配的对象
findRelation(java.lang.String parentType, java.lang.String parentId, java.lang.String relationName)	java.util.List	确定对象是否有子对象
hasChildrenInRelation(java.lang.String parentType, java.lang.String parentId, java.lang.String relationName)	HasChildrenResult	按某种特定关系查找与给定父对象相关的所有子对象
invalidate(java.lang.String type, java.lang.String id)	Void	通过插件按类型和 ID 使对象无效

IPluginNotificationHandler 接口

IPluginNotificationHandler 可定义用于向 Orchestrator 通知它通过插件访问的对象上发生的各种类型的事件的方法。

IPluginNotificationHandler 接口可定义以下方法。

方法	返回	描述
getSessionID()	java.lang.String	返回当前会话 ID
notifyElementDeleted(java.lang.String type, java.lang.String id)	Void	通知系统，具有给定类型和 ID 的对象已 删除

方法	返回	描述
notifyElementInvalidate(java.lang.String type, java.lang.String id)	Void	通知系统，对象的关系已更改
notifyElementUpdated(java.lang.String type, java.lang.String id)	Void	通知系统，对象的属性已修改
notifyMessage(ch.dunes.vso.sdk.api.ErrorLevel severity, java.lang.String type, java.lang.String id, java.lang.String message)	Void	发布与当前模块相关的错误消息

IPluginPublisher 接口

IPluginPublisher 接口可为要监视的长时间运行的工作流 Wait Event 元素在事件通知总线上发布一个监视程序事件。

IPluginPublisher 接口可定义以下方法。

类型	值	描述
pushWatcherEvent(java.lang.String id, java.util.Properties properties)	Void	在事件通知总线上发布一个监视程序事件

PluginExecutionException 类

如果插件在运行时遇到异常，PluginExecutionException 类将返回一条错误消息。

```
public class PluginExecutionException
extends java.lang.Exception
implements java.io.Serializable
```

- 构造函数：PluginExecutionException(java.lang.String message)
- PluginExecutionException 类从 class java.lang.Throwable 继承以下方法：

  fillInStackTrace、getCause、getLocalizedMessage、getMessage、getStackTrace、initCause、printStackTrace、printStackTrace、printStackTrace、setStackTrace、toStringfillInStackTrace、getCause、getLocalizedMessage、getMessage、getStackTrace、initCause、printStackTrace

- PluginExecutionException 类从 class java.lang.Object 继承以下方法：

  clone、equals、finalize、getClass、hashCode、notify、notifyAll、wait、wait、wait

PluginOperationException 类

PluginOperationException 类可处理插件运行期间遇到的错误。

```
public class PluginOperationException
extends java.lang.RuntimeException
implements java.io.Serializable
```

- 构造函数：PluginOperationException(java.lang.String message)
- PluginOperationException 类从 class java.lang.Throwable 继承以下方法：

  fillInStackTrace、getCause、getLocalizedMessage、getMessage、getStackTrace、initCause、printStackTrace、printStackTrace、printStackTrace、setStackTrace、toString

- PluginOperationException 类从 class java.lang.Object 继承以下方法：

  clone、equals、finalize、getClass、hashCode、notify、notifyAll、,wait、wait、wait

PluginTrigger 类

`PluginTrigger` 类代表长时间运行的工作流 Wait Event 元素定义触发器模块，该模块用于获取有关插件技术中的事件触发器的信息。

```
public class PluginTrigger
extends java.lang.Object
implements java.io.Serializable
```

`PluginTrigger` 类可定义以下方法：

方法	返回	描述
`getModuleName()`	`java.lang.String`	获取触发器模块
`getProperties()`	`java.util.Properties`	获取触发器属性
`getSdkId()`	`java.lang.String`	获取构建模块的 SDK 的 ID
`getSdkType()`	`java.lang.String`	获取构建模块的 SDK 的类型
`getTimeout()`	Long	获取触发器超时时长
`setModuleName(java.lang.String moduleName)`	Void	设置触发器模块名称
`setProperties(java.util.Properties properties)`	Void	设置触发器模块属性
`setSdkId(java.lang.String sdkId)`	Void	设置触发器 SDK 的 ID
`setSdkType(java.lang.String sdkType)`	Void	设置触发器 SDK 的类型
`setTimeout(long timeout)`	Void	设置触发器模块超时

- 构造函数：
  - `PluginTrigger()`
  - `PluginTrigger(java.lang.String moduleName, long timeout, java.lang.String sdkType, java.lang.String sdkId)`
- `PluginTrigger` 类从 `class java.lang.Object` 继承以下方法：

  `clone`、`equals`、`finalize`、`getClass`、`hashCode`、`notify`、`notifyAll`、`toString`、`wait`、`wait`、`wait`

PluginWatcher 类

`PluginWatcher` 类代表长时间运行的工作流 Wait Event 元素监视插件技术中已定义事件的触发器模块。

```
public class PluginWatcher
extends java.lang.Object
implements java.io.Serializable
```

`PluginWatcher` 类可定义以下方法：

方法	返回	描述
`getId()`	`java.lang.String`	获取触发器的 ID
`getModuleName()`	`java.lang.String`	获取触发器模块的名称
`getTimeoutDate()`	Long	获取触发器的超时日期
`getTrigger()`	Void	获取触发器

方法	返回	描述
`setId(java.lang.String id)`	Void	设置触发器的 ID
`setTimeoutDate()`	Void	设置触发器的超时日期

- 构造函数：`PluginWatcher(PluginTrigger trigger)`
- `PluginWatcher` 类从 `class java.lang.Object` 继承以下方法：

  `clone`、`equals`、`finalize`、`getClass`、`hashCode`、`notify`、`notifyAll`、`toString`、`wait`、`wait`、`wait`

QueryResult 类

`QueryResult` 类包含对 Orchestrator 通过插件访问的对象进行的 `find` 查询的结果。

```
public class QueryResult
extends java.lang.Object
implements java.io.Serializable
```

如果找到的结果总数大于查询返回的结果数，则 `totalCount` 的值可能大于 `QueryResult` 返回的元素数。查询返回的结果数在 `vso.xml` 文件的查询语法中定义。

`QueryResult` 类可定义以下方法：

方法	返回	描述
`addElement(java.lang.Object element)`	Void	将元素添加到 `QueryResult`
`addElements(java.util.List elements)`	Void	将元素列表添加到 `QueryResult`
`getElements()`	`java.util.List`	从插入应用程序中获取元素
`getTotalCount()`	Long	获取插件技术中所有可用元素的总数
`isPartialResult()`	Boolean	确定获取的结果是否完整
`removeElement(java.lang.Object element)`	Void	从插件技术中移除元素
`setElements(java.util.List elements)`	Void	设置插件技术中的元素
`setTotalCount(long totalCount)`	Void	设置插件技术中可用元素的总数

- 构造函数：
  - `QueryResult()`
  - `QueryResult(java.util.List ret)`

    `totalCount = ret.size()`
  - `QueryResult(java.util.List elements, long totalCount)`
- `QueryResult` 类从 `class java.lang.Object` 继承以下方法：

  `clone`、`equals`、`finalize`、`getClass`、`hashCode`、`notify`、`notifyAll`、`toString,wait`、`wait`、`wait`

SDKFinderProperty 类

`SDKFinderProperty` 类可定义对由 Orchestrator 查找程序对象在插件技术中找到的对象获取和设置属性的方法。`IDynanmicFinder.getProperties` 方法返回 `SDKFinderProperty` 对象。

```
public class SDKFinderProperty
extends java.lang.Object
```

`SDKFinderProperty` 类可定义以下方法：

方法	返回	描述
getAttributeName()	java.lang.String	获取对象属性名称
getBeanProperty()	java.lang.String	从 Java bean 中获取属性
getDescription()	java.lang.String	获取对象描述
getDisplayName()	java.lang.String	获取对象显示名称
getPossibleResultType()	java.lang.String	获取查找程序可能返回的结果类型
getPropertyAccessor()	java.lang.String	获取对象属性访问器
getPropertyAccessorTree()	java.lang.Object	获取对象属性访问器树
isHidden()	Boolean	显示或隐藏对象
isShowInColumn()	Boolean	显示或隐藏数据库列中的对象
isShowInDescription()	Boolean	显示或隐藏对象描述
setAttributeName(java.lang.String attributeName)	Void	设置对象属性名称
setBeanProperty(java.lang.String beanProperty)	Void	在 Java bean 中设置属性
setDescription(java.lang.String description)	Void	设置对象描述
setDisplayName(java.lang.String displayName)	Void	设置对象显示名称
setHidden(boolean hidden)	Void	显示或隐藏对象
setPossibleResultType(java.lang.String possibleResultType)	Void	设置查找程序可能返回的结果类型
setPropertyAccessor(java.lang.String propertyAccessor)	Void	设置对象属性访问器
setPropertyAccessorTree(java.lang.Object propertyAccessorTree)	Void	设置对象属性访问器树
setShowInColumn(boolean showInTable)	Void	显示或隐藏数据库列中的对象
setShowInDescription(boolean showInDescription)	Void	显示或隐藏对象描述

- 构造函数：SDKFinderProperty(java.lang.String attributeName, java.lang.String displayName, java.lang.String beanProperty, java.lang.String propertyAccessor)
- SDKFinderProperty 类从 class java.lang.Object 继承以下方法：

  clone、equals、finalize、getClass、hashCode、notify、notifyAll、toString、wait、wait、wait

HasChildrenResult 枚举

HasChildrenResult 枚举声明给定父对象是否有子对象。IPluginFactory.hasChildrenInRelation 方法返回 HasChildrenResult 对象。

```
public enum HasChildrenResult
extends java.lang.Enum<HasChildrenResult>
implements java.io.Serializable
```

HasChildrenResult 枚举可定义以下常量：

- public static final HasChildrenResult Yes
- public static final HasChildrenResult No
- public static final HasChildrenResult Unknown

HasChildrenResult 枚举可定义以下方法：

方法	返回	描述
getValue()	int	返回以下值之一：HasChildrenResult **1** 父对象有子对象 **-1** 父对象没有子对象 **0** 未知，或无效参数
valueOf(java.lang.String name)	static HasChildrenResult	返回一个具有指定名称的此类型的枚举常量。字符串必须与用于声明此类型枚举常量的标识符完全匹配。不要在枚举名称中使用空白字符。
values()	static HasChildrenResult[]	返回包含此枚举类型的常量的数组，以便进行声明。此方法可按如下方法迭代 常量： `for (HasChildrenResult c : HasChildrenResult.values()) System.out.println(c);`

- HasChildrenResult 枚举从 class java.lang.Enum 继承以下方法：

  clone、compareTo、equals、finalize、getDeclaringClass、hashCode、name、ordinal、toString、valueOf

- HasChildrenResult 枚举从 class java.lang.Object 继承以下方法：

  getClass、notify、notifyAll、wait、wait、wait

ScriptingAttribute 注释类型

ScriptingAttribute 注释类型为插件技术中的对象属性提供注释，以用作脚本属性。

```
@Retention(value=RUNTIME)
@Target(value={METHOD,FIELD})
public @interface ScriptingAttribute
```

ScriptingAttribute 注释类型具有以下值：

```
public abstract java.lang.String value
```

ScriptingFunction 注释类型

ScriptingFunction 注释类型为方法提供注释，以用作脚本属性。

```
@Retention(value=RUNTIME)
@Target(value={METHOD,CONSTRUCTOR})
public @interface ScriptingFunction
```

ScriptingFunction 注释类型具有以下值：

```
public abstract java.lang.String value
```

ScriptingParameter 注释类型

ScriptingParameter 注释类型为参数提供注释，以用作脚本属性。

```
@Retention(value=RUNTIME)
@Target(value=PARAMETER)
public @interface ScriptingParameter
```

ScriptingParameter 注释类型具有以下值：

```
public abstract java.lang.String value
```

开发 Web 服务客户端 7

VMware vCenter Orchestrator 提供 Web 服务 API，以便您可以开发应用程序以通过 Web 服务访问工作流。Orchestrator Web 服务主要用于启动工作流以及通过网络或 Web 检索工作流的输出参数。

Web 服务 API 可提供一组对象，或者是 Web 服务定义语言 (WSDL) 类型定义和一组方法或 Web 服务操作，通过它们可以获取工作流、运行工作流、刷新工作流状态以及获取其输出参数值。此外，Web 服务 API 还可基于从插件获取的对象之间的关系实现树查看器功能。此 API 中的复杂对象类型并不多，而且操作也相对较少。

注意 为了帮助了解 Orchestrator 如何实现 Web 服务，请先熟悉您的开发框架的 Web 服务 API，例如 Java 或 .Net。

本章讨论了以下主题：

编写 Web 服务客户端应用程序

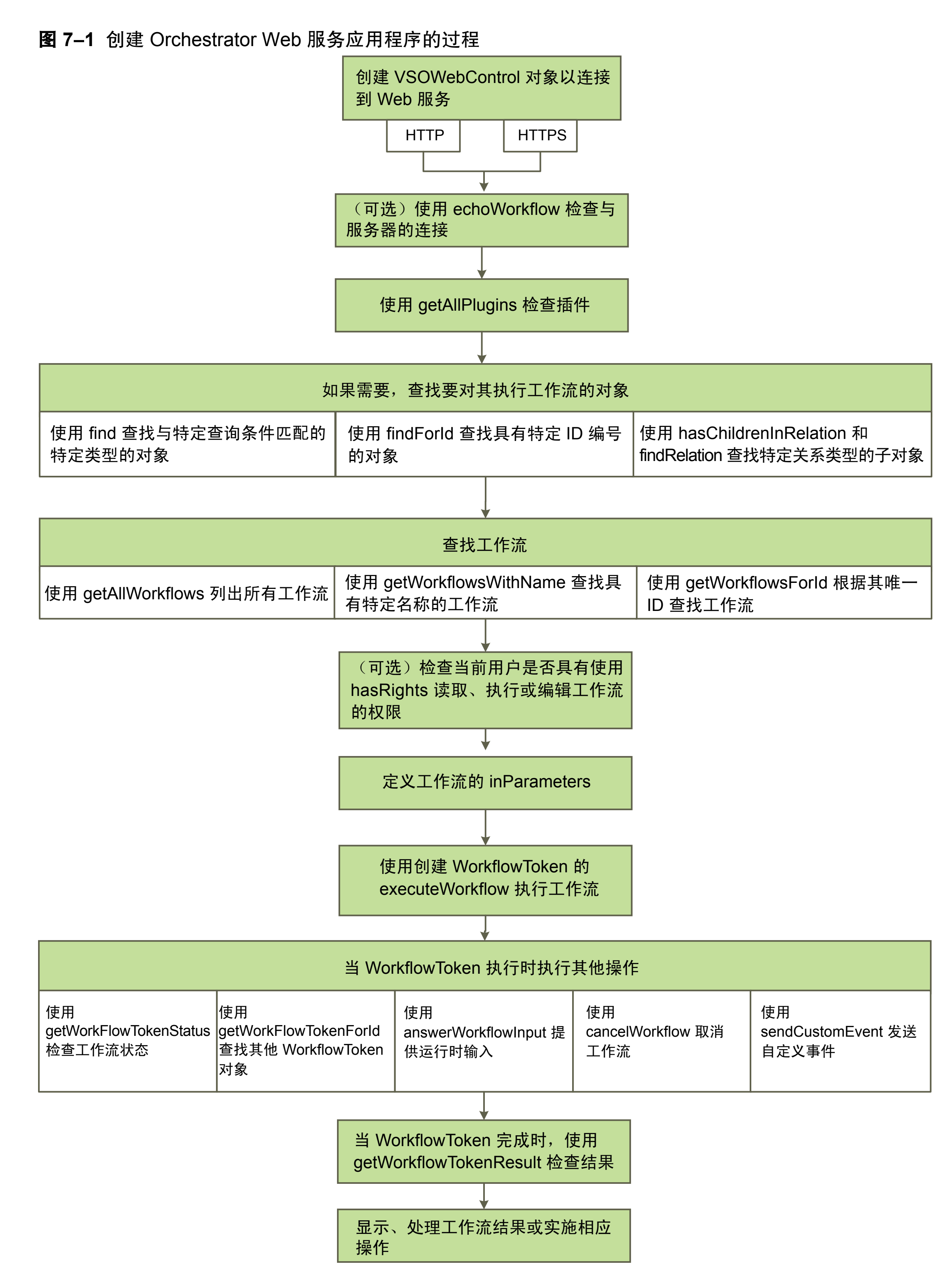
使用 Orchestrator Web 服务 API 的大多数应用程序都具有共同的结构。要创建 Orchestrator Web 服务客户端应用程序，必须按顺序执行一系列标准任务。

创建 Orchestrator Web 服务客户端应用程序的过程

开发 Web 服务客户端应用程序时需按顺序经历多个开发阶段。

图 7‑1 显示了如何创建典型的 Orchestrator Web 服务客户端应用程序。

图 7–1 创建 Orchestrator Web 服务应用程序的过程



请按照图示的各个开发阶段来创建可满足您的大多数要求的 Orchestrator Web 服务客户端应用程序。

Web 服务端点

Web 服务端点是一种可用于将 Web 服务客户端连接到 Orchestrator 服务器的端口。

连接到位于下列 URL 的 Orchestrator Web 服务端点，其中 <*orchestrator_server*> 是 Orchestrator 服务器在其上运行的主机的 IP 地址。

```
http://<orchestrator_server>:8280/vmware-vmo-webcontrol/webservice
```

Web 服务可通过 HTTP 或 HTTPS 在 Orchestrator 服务器的端口 8280 或 8281 上运行。访问 Web 服务 API 需要在 Orchestrator 服务器上拥有有效的用户名和密码。因为每次访问该服务时都必须单独进行身份验证，所以安全 HTTPS 连接不是必需的。但是，Web 服务通过网络发送密码时并未进行加密，所以如果您的应用程序需要保证安全，请使用安全 HTTPS 连接。

注意 通过 HTTPS 保护的网络将会访问位于端口 8281 上的 Web 服务端点。在您的网络中，此端口号可能与默认的 8280 或 8281 有所不同。

生成 Orchestrator Web 服务存根

可以通过 Orchestrator WSDL 描述文件生成 Web 服务对象，以此为 Web 服务应用程序创建客户端和服务器存根。

Orchestrator 将在以下位置发布 WSDL 描述文件。

```
http://<orchestrator_server>:8280/vmware-vmo-webcontrol/webservice?WSDL
```

通过使用 Java 或 .Net 代码生成器可以生成 Web 服务客户端和服务器存根。Orchestrator Web 服务可支持所有 WSDL 1.1 分析程序。通过生成 Web 服务可提供以下对象。

VSOWebControl
Web 服务将定义一个名为 `VSOWebControl` 的 WSDL 端口类型，通过此端口可访问所有的 Orchestrator Web 服务操作。

WebServiceStub
Web 服务将定义应用程序用于启动 Web 服务的客户端和服务器端存根。

VSOWebControlProxy
Web 服务通过代理提供对 Orchestrator Web 服务操作的访问。

VSOWebControlService
`VSOWebControlService` 服务是一种远程过程调用 (RPC) `Service` 实施。

VSOWebControlServiceLocator
`VSOWebControlServiceLocator` 服务可扩展 `VSOWebControlService`，从而提供以下操作。

- `getwebserviceAddress`，可用于获取 Web 服务的端点 URL。
- `getwebservice`，可用于获取 Web 服务应用程序的客户端存根，并可通过相应的端点 URL 实例化 `VSOWebControl` 端口类型对象。

创建 Web 服务客户端

可以通过 Orchestrator Web 服务 API 创建 Web 服务客户端，以连接到 Orchestrator 服务器。通过 Web 服务连接，您可以在 Orchestrator 服务器中访问工作流并对其执行操作。

前提条件

必须已使用代码生成器根据 Orchestrator WSDL 定义生成了 Web 服务客户端和服务器存根。

步骤

1 连接到 Orchestrator Web 服务第 150 页，

Web 服务应用程序使用 HTTP 或 HTTPS 协议，通过简单对象访问协议 (SOAP) 绑定建立与 Orchestrator 服务器的连接。

2 在 Orchestrator 服务器中查找对象第 151 页，

要通过工作流执行任何有用的任务，必须找到该工作流将在其上运行的对象。Orchestrator Web 服务 API 提供了一些可用于在 VMware Infrastructure 清单中查找所有对象类型的函数。

3 通过使用 find 操作查找对象第 152 页，

可以使用 `find` 操作查找匹配特定搜索条件（在 `query` 参数中设置）的任何类型的对象。

4 通过使用 findForId 操作查找对象第 153 页，

如果知道某个特定对象的唯一 ID，则可以使用 `findForId` 操作查找该对象。

5 通过使用 findRelation 操作查找对象第 154 页，

可以使用 `findRelation` 操作找到特定对象的子对象。

6 在 Orchestrator 服务器中查找工作流第 155 页，

当找到要与之交互的对象后，必须查找执行这些交互操作的工作流。

7 通过使用 getAllWorkflows 操作查找工作流第 156 页，

`getAllWorkflows` 操作列出了用户可作为 `Workflow` 对象数组访问的所有工作流。

8 通过使用 getWorkflowsWithName 操作查找工作流第 156 页，

如果已知某个特定工作流的名称（如在 Orchestrator 客户端中定义的名称），则 Web 服务应用程序可以使用其名称或名称的一部分获取此工作流。

9 通过使用 getWorkflowForID 操作查找工作流第 156 页，

如果已知某个特定工作流 ID，则 Web 服务应用程序可以通过使用 `getWorkflowForID` 操作获取此工作流。

10 从 Web 服务客户端运行工作流第 157 页，

Web 服务客户端的主要用途是通过网络运行工作流。

11 在工作流运行时与其交互第 158 页，

工作流启动之后，Web 服务客户端可为响应工作流运行时发生的事件而执行各种操作。

12 获取工作流结果第 160 页，

工作流运行完毕之后，可通过调用 `getWorkflowTokenResult( )` 操作检索结果。

连接到 Orchestrator Web 服务

Web 服务应用程序使用 HTTP 或 HTTPS 协议，通过简单对象访问协议 (SOAP) 绑定建立与 Orchestrator 服务器的连接。

前提条件

必须已根据 Orchestrator WSDL 定义生成了 Orchestrator Web 服务客户端和服务器存根。此外，还必须创建一个可实现 `VSOWebControl` 接口的 Web 服务客户端应用程序类。

步骤

1 在 Web 服务客户端应用程序类中，创建一个可连接到 Web 服务端点的 `VSOWebControl` 实例。

可以使用HTTP创建非安全连接，或者也可以使用HTTPS创建安全连接。默认的HTTP端口是8280，默认的HTTPS 端口是 8281。URL 也是默认的。

- 以下示例显示如何创建与 Web 服务的 HTTP 连接。

```
String urlprefix = "http://10.0.0.1:8280" ;
URL url = new URL(urlprefix + "/vmware-vmo-webcontrol/webservice");
vsoWebControl = new VSOWebControlServiceLocator().getwebservice(url);
```

- 以下示例显示如何创建与 Web 服务的 HTTPS 连接。

```
String urlprefix = "https://10.0.0.1:8281" ;
URL url = new URL(urlprefix + "/vmware-vmo-webcontrol/webservice");
vsoWebControl = new VSOWebControlServiceLocator().getwebservice(url);
```

2 通过调用 `echo` 操作，检查服务器连接。

以下示例显示如何调用 `echo` 操作。

```
vsoWebControl.echo(string);
```

对 `echo` 操作的上述调用将返回作为参数提供的字符串对象。

3 （可选）通过调用 `getAllPlugins` 操作，检查哪些插件正在运行 Orchestrator 服务器。

以下示例显示如何调用 `getAllPlugins` 操作。

```
ModuleInfo[] modules = vsoWebControl.getAllPlugins(username, password);
```

对 `getAllPlugins` 操作的上述调用将返回一个 `ModuleInfo` 对象数组，每个对象均包含有关在 Orchestrator 服务器中运行的一个插件的名称和版本信息。

您已创建与 Orchestrator Web 服务的连接，验证了该连接，并且建立了插入到 Orchestrator 服务器中的技术。

下一步

通过 Web 服务连接在 Orchestrator 服务器中查找对象。

在 Orchestrator 服务器中查找对象

要通过工作流执行任何有用的任务，必须找到该工作流将在其上运行的对象。Orchestrator Web 服务 API 提供了一些可用于在 VMware Infrastructure 清单中查找所有对象类型的函数。

工作流通常在 vCenter Server 中的对象上运行。此外，工作流也可以在 vCenter Server 以外的对象上运行，方法是通过插件访问这些对象。

Web 服务 API 定义的用于查找对象的操作如下。

- `find`
- `findForId`
- `findRelation`
- `hasChildrenInRelation`

所有用于查找对象的操作均会返回 `FinderResult` 对象，这些对象可单独返回，以数组形式返回，或者嵌入在 `QueryResult` 对象中返回。

通过使用 find 操作查找对象

可以使用 `find` 操作查找匹配特定搜索条件（在 `query` 参数中设置）的任何类型的对象。

您通过其访问对象的插件的 `vso.xml` 文件定义了 `query` 参数的语法。

前提条件

必须已经在 Web 服务客户端应用程序类中创建了与 Orchestrator Web 服务端点的连接。

步骤

1 通过在对象上调用 `find` 操作创建一个 `QueryResult` 对象。

以下代码示例显示应用程序如何调用 `find` 操作，以便了解特定用户可通过 vCenter Server 4.0 插件访问的虚拟机的数量。

```
QueryResult queryResult = vsoWebControl.find("VC:VirtualMachine", null,
<用户名>, <密码>);
if (queryResult != null) {
System.out.println("Found " + queryResult.getTotalCount() +
" objs.");
FinderResult[] elts = queryResult.getElements();
finderResult = elts[0];
displayFinderResult(finderResult);
            }
else {
System.out.println("Found nothing");
            }
```

根据 vCenter Server 4.0 插件定义的查询语法，将 `query` 参数设置为 `null` 可返回由第一个参数指定的类型的所有对象的列表。上述代码示例将执行以下任务。

- 获取库中所有 `VC:VirtualMachine` 对象的列表。
- 调用 `QueryResult` 对象的 `getTotalCount` 操作，以获取找到的 `VC:VirtualMachine` 对象的总数并打印该值。

- 调用 QueryResult 对象的 getElements 操作，以获取作为 FinderResult 对象数组找到的对象的详细信息。
- 将 FinderResult 对象数组传递到内部方法 displayFinderResult，从而提取相关信息。

2 从 FinderResult 对象提取结果。

   要显示、解释或处理由 find 操作返回的 FinderResult 对象中的结果，必须将这些结果传送到 Web 服务应用程序。

   以下示例显示如何提取 FinderResult 对象中返回的结果。

```
public static void displayFinderResult(FinderResult finderResult) {
if (finderResult != null) {
System.out.println("Finder result is of type '"
+ finderResult.getType()
+ "', id '" + finderResult.getId()
+ "' and uri '"
+ finderResult.getDunesUri() + "'");
System.out.println("And has properties :");
Property[] props = finderResult.getProperties();
if (props != null) {
for (int ii = 0; ii < props.length; ii++) {
System.out.println("\t" + props[ii].getName() + "="
+ props[ii].getValue());
      }
    }
  }
```

   该示例定义了一种内部方法 displayFinderResult，该方法将选取一个 FinderResult 对象，并获取和显示其类型、ID、所在的 URI 及其属性。例如，您可以使用 URI 调用工作流。getType、getId、getProperties 和 getDunesUri 方法均由 FinderResult 对象定义。

您已在 Orchestrator 服务器中找到 Web 服务客户端可访问并在其上运行工作流的对象。

下一步

在客户端应用程序中实施 Web 服务操作以在 Orchestrator 服务器中查找工作流。

通过使用 findForId 操作查找对象

如果知道某个特定对象的唯一 ID，则可以使用 findForId 操作查找该对象。

要使用 findForId，必须将特定类型的对象与其标识符匹配。

前提条件

必须已经在 Web 服务客户端应用程序类中创建了与 Orchestrator Web 服务端点的连接。

步骤

1 通过在对象上调用 `findForId` 操作，创建一个 `FinderResult` 对象。

```
finderResult = vsoWebControl.findForId("VC:VirtualMachine", "vcenter/vm-xx",
username, password);
```

在上述示例中，`vcenter/vm-xx` 是 `findForID` 操作查找的虚拟机对象的 ID。

`findForID` 操作直接返回一个 `FinderResult` 实例，而不会像 `find` 一样创建一个 `FinderResult` 对象数组。按对象的唯一 ID 查找对象始终只会返回一个对象。

2 从 `FinderResult` 对象提取结果。

要显示、解释或处理由 `find` 操作返回的 `FinderResult` 对象中的结果，必须将这些结果传送到 Web 服务应用程序。

以下示例显示如何提取 `FinderResult` 对象中返回的结果。

```
public static void displayFinderResult(FinderResult finderResult) {
if (finderResult != null) {
System.out.println("Finder result is of type '"
+ finderResult.getType()
+ "', id '" + finderResult.getId()
+ "' and uri '"
+ finderResult.getDunesUri() + "'");
System.out.println("And has properties :");
Property[] props = finderResult.getProperties();
if (props != null) {
for (int ii = 0; ii < props.length; ii++) {
System.out.println("\t" + props[ii].getName() + "="
+ props[ii].getValue());
      }
    }
  }
```

该示例定义了一种内部方法 `displayFinderResult`，该方法将选取一个 `FinderResult` 对象，并获取和显示其类型、ID、所在的 URI 及其属性。例如，您可以使用 URI 调用工作流。`getType`、`getId`、`getProperties` 和 `getDunesUri` 方法均由 `FinderResult` 对象定义。

您已在 Orchestrator 服务器中找到 Web 服务客户端可访问并在其上运行工作流的对象。

下一步

在客户端应用程序中实施 Web 服务操作以在 Orchestrator 服务器中查找工作流。

通过使用 findRelation 操作查找对象

可以使用 `findRelation` 操作找到特定对象的子对象。

`findRelation` 操作将返回 `FinderResult` 对象数组，该数组与某个特定对象的子对象相对应。

前提条件

必须已经在 Web 服务客户端应用程序类中创建了与 Orchestrator Web 服务端点的连接。

步骤

1 通过在某个对象上调用 `findRelation` 操作，创建一个 `FinderResult` 对象数组。

```
FinderResult[] results = vsoWebControl.findRelation("VC:ComputeResource",
"vcenter/domain-s114", "getResourcePool()", "username", "password");
```

以上示例可返回匹配以下条件的 `FinderResult` 对象数组。

- 父元素的类型为 `VC:ComputeResource`。
- 父元素的 ID 为 `vchost/domain-s114`。
- 返回的子元素通过 Orchestrator vCenter Server 4 插件定义的 `getResourcePool` 关系与父元素相关。

2 从 `FinderResult` 对象提取结果。

要显示、解释或处理由 `find` 操作返回的 `FinderResult` 对象中的结果，必须将这些结果传送到 Web 服务应用程序。

以下示例显示如何提取 `FinderResult` 对象中返回的结果。

```
public static void displayFinderResult(FinderResult finderResult) {
if (finderResult != null) {
System.out.println("Finder result is of type '"
+ finderResult.getType()
+ "', id '" + finderResult.getId()
+ "' and uri '"
+ finderResult.getDunesUri() + "'");
System.out.println("And has properties :");
Property[] props = finderResult.getProperties();
if (props != null) {
for (int ii = 0; ii < props.length; ii++) {
System.out.println("\t" + props[ii].getName() + "="
+ props[ii].getValue());
      }
    }
  }
```

该示例定义了一种内部方法 `displayFinderResult`，该方法将选取一个 `FinderResult` 对象，并获取和显示其类型、ID、所在的 URI 及其属性。例如，您可以使用 URI 调用工作流。`getType`、`getId`、`getProperties` 和 `getDunesUri` 方法均由 `FinderResult` 对象定义。

您已在 Orchestrator 服务器中找到 Web 服务客户端可访问并在其上运行工作流的对象。

下一步

在客户端应用程序中实施 Web 服务操作以在 Orchestrator 服务器中查找工作流。

在 Orchestrator 服务器中查找工作流

当找到要与之交互的对象后，必须查找执行这些交互操作的工作流。

Orchestrator Web 服务 API 包含以下操作，可用于查找在给定环境中运行的所有工作流、具有特定名称的工作流或具有特定 ID 的工作流。

- `getAllWorkflows`
- `getWorkflowsWithName`
- `getWorkflowForID`

通过使用 getAllWorkflows 操作查找工作流

getAllWorkflows 操作列出了用户可作为 Workflow 对象数组访问的所有工作流。

因为 getAllWorkflows 操作可返回包含与工作流相关的所有信息的 Workflow 对象，所以这对需要完整的工作流信息（如工作流的名称、ID、描述、参数和属性）的应用程序很有帮助。

前提条件

必须已经在客户端应用程序中实施了在 Orchestrator 服务器中查找对象的 Web 服务操作。

步骤

◆ 通过调用 getAllWorkflows 操作，创建一个 Workflow 对象数组。

```
Workflow[] workflows = vsoWebControl.getAllWorkflows(username, password);
```

以上代码示例通过调用 getAllWorkflows 获取 Web 服务客户端可以运行的 Workflow 对象数组。

您已在 Orchestrator 服务器中找到 Web 服务客户端可在对象上运行的工作流。

下一步

在 Web 服务客户端中实施相应操作以运行找到的工作流。

通过使用 getWorkflowsWithName 操作查找工作流

如果已知某个特定工作流的名称（如在 Orchestrator 客户端中定义的名称），则 Web 服务应用程序可以使用其名称或名称的一部分获取此工作流。

getWorkflowsWithName 操作将返回一个工作流数组，以便您使用该数组通过通配符匹配多个工作流。

前提条件

必须已经在客户端应用程序中实施了在 Orchestrator 服务器中查找对象的 Web 服务操作。

步骤

◆ 通过调用 getWorkflowsWithName 操作，创建一个 Workflow 对象数组。

```
Workflow[] workflows =
vsoWebControl.getWorkflowsWithName("Simple user interaction",
username, password);
```

以上代码示例将通过调用 getWorkflowsWithName 操作获取所有名称或名称的一部分为 Simple user interaction 的工作流。

您已在 Orchestrator 服务器中找到 Web 服务客户端可在对象上运行的工作流。

下一步

在 Web 服务客户端中实施相应操作以运行找到的工作流。

通过使用 getWorkflowForID 操作查找工作流

如果已知某个特定工作流 ID，则 Web 服务应用程序可以通过使用 getWorkflowForID 操作获取此工作流。

因为所有工作流 ID 都是唯一的，所以 getWorkflowForID 操作将返回一个 Workflow 实例。

前提条件

必须已经在客户端应用程序中实施了在 Orchestrator 服务器中查找对象的 Web 服务操作。

步骤

◆ 通过调用 `getWorkflowForID` 操作，创建一个 `Workflow` 对象。

```
String workflowId = "1880808080808080808080808080808087808080011713796199469943be4c882";
Workflow workflow = vsoWebControl.getWorkflowForID(workflowId, username, password);
```

您已在 Orchestrator 服务器中找到 Web 服务客户端可在对象上运行的一个工作流。

下一步

在 Web 服务客户端中实施相应操作以运行找到的工作流。

从 Web 服务客户端运行工作流

Web 服务客户端的主要用途是通过网络运行工作流。

前提条件

必须已在客户端中实施了在 Orchestrator 服务器中查找工作流的 Web 服务操作。

步骤

1 （可选）通过调用 hasRights 操作，检查工作流用户权限。

可以使用 hasRights 操作验证用户是否有权读取、运行或编辑某个特定工作流。此操作并非强制性操作，但在运行工作流之前检查用户权限有助于防止异常情况发生。

```
String workflowId = "188080808080808080808080808080808780808001171379619946994 3be4c882";
Boolean rights = vsoWebControl.hasRights(workflowId, username, password, 'x');
```

以上代码示例通过调用 hasRights 操作来发现用户是否有权运行由 workflowId 标识的工作流。

如果用户有权运行该工作流，hasRights 将返回 true。否则，hasRights 会返回 false。

2 在 WorkflowTokenAttribute 对象中设置工作流属性。

Web 服务客户端会将 WorkflowTokenAttributes 数组传递给用于运行工作流的 WorkflowToken 对象。

```
WorkflowTokenAttribute[] attributes = new WorkflowTokenAttribute[1];
WorkflowTokenAttribute attribute = new WorkflowTokenAttribute();
attribute.setName("vm");
attribute.setType(finderResult.getType());
attribute.setValue(finderResult.getDunesUri());
attributes[0] = attribute;
```

以上示例将创建一个 WorkflowTokenAttribute 对象，然后使用以下信息对其进行填充：

- 属性名称，在此示例中为 vm。
- 属性类型，与在本段代码的其他位置定义的 FinderResult 对象中发现的类型一样。
- 属性值，在此示例中为 dunesUri 字符串，表示该值指明一个可通过插件访问的对象。

3 通过调用 executeWorkflow 操作，运行该工作流。

要运行工作流，则必须以 WorkflowTokenAttribute 数组的形式将工作流属性传递到 executeWorkflow 操作中。

通过运行工作流可创建一个 WorkflowToken 对象，该对象表示工作流启动时使用收到的特定输入参数运行的工作流实例。

```
WorkflowToken token = vsoWebControl.executeWorkflow(workflowId, username, password,
attributes);
```

在以上示例中，attributes 属性是在步骤 2 中创建的 WorkflowTokenAttribute 对象的数组。

有时，工作流在其运行期间需要输入参数。在此类情况下，可通过用户交互在工作流运行的同时提供属性。通过使用 answerWorkflowInput 操作，可在工作流运行期间将属性传递到工作流中。

您已在 Web 服务客户端中实施了检查用户权限、将属性传递到工作流以及运行工作流的操作。

下一步

当工作流运行时，在 Web 服务客户端中实施相应操作以与工作流交互。

在工作流运行时与其交互

工作流启动之后，Web 服务客户端可为响应工作流运行时发生的事件而执行各种操作。

前提条件

必须已经在 Web 服务客户端中实施了在 Orchestrator 服务器中运行工作流的操作。

步骤

1 通过调用 getWorkflowTokenForId 操作，查找正在运行的工作流。

调用 getWorkflowTokenForId 可以获取 WorkflowToken 对象，此对象包含有关该特定工作流令牌的所有信息。

```
WorkflowToken onemoretoken = vsoWebControl.getWorkflowTokenForId(workflowTokenId, username,
password);
AllActiveWorkflowTokens[n] = onemoretoken;
```

以上代码示例通过对象 ID 获取一个 WorkflowToken 对象，并且将其设置为正在运行的 WorkflowToken 对象数组中的一员。

2 通过调用 getWorkFlowTokenStatus 操作，检查工作流令牌的状态。

当工作流运行时，应用程序的主要事件循环通常会以固定的时间间隔全力检查工作流的状态。getWorkflowTokenStatus 操作需要它要为其获取状态的工作流令牌 ID 的数组。

```
String workflowId = workflows[0].getId();
WorkflowToken token = vsoWebControl.executeWorkflow(workflowId, username, password, null);
String[] tokenIds = { token.getId() };
String tokenStatus = "";
while ("completed".equals(tokenStatus) == false
&& "failed".equals(tokenStatus) == false
&& "canceled".equals(tokenStatus) == false
&& "waiting".equals(tokenStatus) == false) { Thread.sleep(1 * 1000);
// Wait 1s
String[] status = vsoWebControl.getWorkflowTokenStatus(tokenIds, username,
password);
tokenStatus = status[0];
System.out.println("Workflow is still running...("+ tokenStatus + ")");
}
```

以上示例将获取工作流令牌数组的 ID。通过在循环中运行 getWorkflowTokenStatus，此操作可同时检查多个 WorkflowToken 对象的状态。

以上示例通过每隔一秒检查一次 WorkflowToken 对象的状态，可保持应用程序及时获得更新。例如，如果工作流处于 waiting 状态，则该工作流正在等待由 answerWorkflowInput 操作提供的运行时输入。

3 通过调用 answerWorkflowInput 操作，提供来自用户交互的输入。

如果工作流正在以 waiting 状态等待用户输入，则应用程序的事件循环可以随时指定该输入。您可以照常创建 WorkflowTokenAttribute 数组，然后在工作流运行期间，通过使用 answerWorkflowInput 操作将这些数组提供给工作流。以下示例继续显示步骤 2 的后续代码。

```
 if ("waiting".equals(tokenStatus) == true) {
System.out.println("Answering user interaction");
WorkflowTokenAttribute[] attributes = new WorkflowTokenAttribute[2];
WorkflowTokenAttribute attribute = null;
attribute = new WorkflowTokenAttribute();
attribute.setName("param1");
attribute.setType("string");
attribute.setValue("answer1");
attributes[0] = attribute;
attribute = new WorkflowTokenAttribute();
attribute.setName("param2");
attribute.setType("number");
attribute.setValue("123");
attributes[1] = attribute;
vsoWebControl.answerWorkflowInput(token.getId(), attributes, username,
password);
            }
```

在以上示例中，如果工作流处于 waiting 状态，则应用程序将创建两个 WorkFlowTokenAttribute 对象。这些对象通过调用各种 WorkFlowTokenAttribute 操作来获取属性值。然后，此过程会将这些 WorkFlowTokenAttribute 对象添加到 WorkflowTokenAttribute 数组中。

4 通过调用 cancelWorkflow 操作可以取消工作流。

使用 cancelWorkflow 操作，随时都能取消工作流。

```
vsoWebControl.cancelWorkflow(workflowTokenId, username, password);
```

5 检查工作流是否已成功取消。

因为 cancelWorkflow 操作不会返回任何内容，所以必须先获取 WorkflowToken 状态，才能确保工作流成功取消，如以下代码示例所示。

```
String[] status = vsoWebControl.getWorkflowTokenStatus(tokenIds, username, password);
if ("canceled".equals(status) == true) {
System.out.println("Workflow canceled");
}
```

Web 服务客户端通过查找工作流的状态、提供来自用户交互的输入参数以及取消工作流来与工作流进行交互。

下一步

在 Web 服务客户端中实施相应操作以提取工作流结果。

获取工作流结果

工作流运行完毕之后，可通过调用 getWorkflowTokenResult( ) 操作检索结果。

前提条件

必须已在 Web 服务客户端中设置了工作流在 Orchestrator 服务器上的启动方式。

步骤

1 通过调用 getWorkflowTokenResult( ) 操作，获取正在运行的工作流的结果。

getWorkflowTokenResult( ) 操作会将这些结果存储为一个属性数组。

```
 WorkflowTokenAttribute[] retAttributes =
vsoWebControl.getWorkflowTokenResult(token.getId(),
username, password);
```

以上示例代码将获取具有特定标识符的工作流令牌的结果。

2 （可选）通过调用 WorkflowTokenAttribute.getValue() 操作，检查工作流结果。

```
WorkflowTokenAttribute resultCode = result[0];
WorkflowTokenAttribute resultMessage = result[1];
System.out.println("Workflow output code ...(" + resultCode.getValue() + ")");
System.out.println("Workflow output message...(" + resultMessage.getValue() + ")");
```

3 动态生成工作流令牌的结果属性以供显示或供其他应用程序使用。

```
for (int ii = 0; ii < retAttributes.length; ii++) {
System.out.println("\tName:'"+ retAttributes[ii].getName()
+ "' - Type:'"+ retAttributes[ii].getType()
+ "' - Value:'"+ retAttributes[ii].getValue()
}
```

以上示例代码将会打印工作流令牌结果属性的名称、类型和值。

您定义了一个可实现以下功能的 Web 服务客户端：在 Orchestrator 中查找对象、在对象上运行工作流、与正在运行的工作流交互以及提取这些工作流的运行结果。

时区与通过 Web 服务运行工作流

如果发出运行请求的应用程序与 Orchestrator 服务器所处的运行时区不同，则通过 Web 服务运行工作流可能会导致时间戳出错。

如果某个工作流将时间和日期作为输入参数，并在运行时将时间和日期作为输出生成，则当此工作流通过 Web 服务应用程序运行时，作为输入参数发送的时间和日期反映的是在其上运行 Web 服务应用程序的系统的时间和日期。作为工作流的输出发送的时间和日期反映的是在其上运行 Orchestrator 服务器的系统的时间和日期。如果 Web 服务应用程序与 Orchestrator 服务器所处的运行时区不同，则该工作流返回的时间将与 Web 服务应用程序调用 executeWorkflow 或 getWorkflowTokenResult 时作为输入提供的时间不匹配。

为避免此问题，您可以创建一个函数，用于比较 Web 服务应用程序中的日期。如果要将时区信息考虑在内，则必须序列化日期和时间。以下 Java 代码示例显示如何将 Orchestrator 返回的字符串转换成 Date 对象。

```
public Date dateFromString(String value){
java.text.DateFormat s_dateFormat = new java.text.SimpleDateFormat("yyyyMMddHHmmssZ");
Date date = null;
if (value != null && value.length() > 0) {
try {
date = s_dateFormat.parse(value);
} catch (ParseException e) {
System.err.println("Converting String to Date :ERROR");
date = null ;
    }
  }
return date;
}
```

Web 服务应用程序示例

Orchestrator 提供多个 Web 服务客户端应用程序的工作示例，这些应用程序可提供对 Orchestrator 的 Web 访问。

您可以从 VMware vCenter Orchestrator 4.0 文档下载页面下载 Orchestrator 示例 ZIP 文件。有关何处可以找到文档下载页面的信息，请参见第 5 页，“示例应用程序”。

Web 服务 API 对象

Orchestrator Web 服务 API 提供一个作为 WSDL 复杂类型使用的对象集合以及一个作为 WSDL 操作使用的方法集合。

FinderResult 对象

`FinderResult` 表示 Orchestrator 可直接或通过插件找到的一个来自 Orchestrator 清单的对象。例如，`FinderResult` 对象可以表示一个来自 vCenter Server 的虚拟机。

`FinderResult` 对象表示插件在其 `vso.xml` 文件中向 Orchestrator 注册的任何对象。`FinderResult` 对象表示当您调用其中一个 `find*` 操作时，从所有已安装的插件中找到的项目。返回的项目可以是虚拟机对象、ESX 对象、基础架构标准或 Orchestrator 工作流令牌。因为大多数工作流都根据 Orchestrator 对象实施操作，所以需要 `FinderResult` 实例作为输入参数。

不能直接将 `FinderResult` 设置为工作流属性。而必须在工作流中设置 `WorkflowTokenAttribute`，此对象将从 `FinderResult` 对象获取类型和 `dunesUri`。

`find` 操作将根据 `vso.xml` 文件中定义的查询条件查找对象。它不会直接返回 `FinderResult` 对象，而是返回 `QueryResult` 对象。`QueryResult` 对象包含多个 `FinderResult` 对象数组。

此外，也可以通过使用 `findForId` 和 `findRelation` 操作，按 ID 或关系来识别所搜索的对象，如以下示例所示。

```
public FinderResult findForId(String type, String id, String username, String password);
public FinderResult[] findRelation(String parentType, String parentId, String relation, String
username, String password);
```

注意 `FinderResult` 不是一个 Orchestrator 可编脚本对象。

下表显示 `FinderResult` 对象的属性。

类型	值	描述
字符串	`type`	找到的对象类型。
字符串	`id`	发现的对象的 ID。
属性数组	`properties`	发现的对象的属性列表。 `properties` 值的格式由每个插件在其 `vso.xml` 文件中定义，位于 `FinderResult` 描述下方。

字符串	`dunesUri`	对象的字符串表示形式。
		如果通过插件访问 `FinderResult` 对象，则该对象将由 `dunesUri` 字符串识别，而不是由另一种类型的字符串或 ID 识别。`dunesUri` 的格式如下。
		`dunes://service.dunes.ch/CustomSDKObject?id='`*<对象_ID>*`'`
		`&dunesName='`*<插件_名称>*`:`*<对象_类型>*`'`

ModuleInfo 对象

ModuleInfo 用于存储每个插件的名称、版本、描述和名称属性。Web 服务应用程序可以根据某些模块或模块版本的存在与否，使用这些属性来修改其行为。

getAllPlugins 操作可返回 ModuleInfo 对象数组，以列出用户可以访问的所有插件，如以下示例所示。

```
public ModuleInfo[] getAllPlugins(username, password);
```

下表显示 ModuleInfo 对象的属性。

类型	值	描述
字符串	moduleName	插件的名称。
字符串	moduleVersion	插件的版本。
字符串	moduleDescription	插件的描述。
字符串	moduleDisplayName	Orchestrator 中显示的插件名称。

Property 对象

Property 对象表示用于描述 Orchestrator 清单项目属性的键/值对。

可通过对 FinderResult 对象调用 getProperties 操作来获取 Property 对象，如以下示例所示。

```
Property[] props = finderResult.getProperties();
```

此方法调用示例将返回 FinderResult 对象的 properties 属性的内容。

下表显示 Property 对象的属性。

类型	值	描述
字符串	name	属性名称。
字符串	value	属性值。 属性值的格式由每个插件在其 vso.xml 文件中定义，位于 FinderResult 描述下方。

QueryResult 对象

QueryResult 对象表示 find 查询的结果。

QueryResult 对象包含一个 FinderResult 对象数组以及一个计数器。QueryResult 对象由 find 操作返回，如以下示例所示。

```
public QueryResult find(String type, String query, String username,
String password);
```

下表显示 QueryResult 对象的属性。

类型	值	描述
Long	totalCount	找到的对象的总数。 QueryResult 对象一个包含 FinderResult 对象数组。相关插件的 vso.xml 文件可设置查询将返回的 FinderResult 对象数量。Orchestrator 提供的标准插件都将返回无限数量的 FinderResult 对象。totalCount 属性报告找到的 FinderResult 对象的总数。如果 totalCount 的值大于插件所设置的数量，则返回的 FinderResults 数组将不会包括查询清单中找到的所有对象。
FinderResult[]	elements	FinderResult 对象数组。

Workflow 对象

`Workflow` 对象表示一种用于定义特定顺序的任务、判定和操作的 Orchestrator 工作流。

具有相应权限的用户可以按名称或按 ID 获取特定的 `Workflow` 对象，或者也可以在服务器中获取有权查看的所有工作流。Orchestrator 提供以下操作以获取 `Workflow` 对象。

```
public Workflow[] getWorkflowsWithName(String name, String username, String password);
public Workflow getWorkflowForId(String workflowId, String username, String password);
public Workflow[] getAllWorkflows(String username, String password);
```

下表显示 `Workflow` 对象的属性。

类型	值	描述
字符串	`id`	工作流 ID。 `id` 字符串是一种全局唯一的 ID 字符串。Orchestrator 创建的工作流拥有标识符，这些标识符是长度极长的字符串，其命名空间发生冲突的可能性非常小。
字符串	`name`	工作流的名称，如 Orchestrator 中工作流的“Name”文本框中所示。
字符串	`description`	工作流具体内容的详细描述。
`WorkflowParameter[]`	`inParameters`	`inParameters` 数组是一组作为工作量输入参数的 `WorkflowParameter` 对象。工作流可以操作这些输入参数，或直接将其用作任务和其他工作流的输入参数。 您可以设置任意输入参数以提供任何需要的输入参数。在运行时省略所需参数会导致工作流失败。
`WorkflowParameter[]`	`outParameters`	`outParameters` 数组是一组因运行工作流而生成的 `WorkflowParameter` 对象。此数组允许工作流发送错误、任何已创建对象的名称以及将其他信息用作输出。 可以设置任意输出参数以生成所需的任何信息。
`WorkflowParameter[]`	`attributes`	`attributes` 数组是一组表示给定工作流的常量和预设变量的 `WorkflowParameter` 对象。属性不同于 `inParameters`，因为属性旨在表示环境常量或变量，而不是运行时信息。 **注意** 不能通过使用 Web 服务检索工作流属性值，只能检索输出参数值。

WorkflowParameter 对象

`WorkflowParameter` 对象可定义工作流中的参数，例如，输入、输出或属性。

工作流开发人员可以设置任意参数，以便为工作流提供所需的任何输入参数或输出参数。参数的格式完全由工作流指定。

下表显示 `WorkflowParameter` 对象的属性。

类型	值	描述
字符串	`name`	参数名称。
字符串	`type`	参数类型。

WorkflowToken 对象

WorkflowToken 对象表示一个处于 running、waiting、waiting-signal、canceled、completed 或 failed 状态的特定工作流实例。

通过启动工作流或通过按其 ID 获取现有工作流令牌，可以获得一个 WorkflowToken 对象，如以下方法签名所示。

```
public WorkflowToken executeWorkflow(String workflowId, String username, String password,
WorkflowTokenAttribute[] attributes);
public WorkflowToken getWorkflowTokenForId(String workflowTokenId, String username, String
password);
```

下表显示 WorkflowToken 对象的属性。

类型	值	描述
字符串	id	这个特定的已完成的工作流的标识符。
字符串	title	这个特定的已完成的工作流的标题。 尽管在启动工作流时某些操作确实允许您设置不同的WorkflowToken标题，但默认情况下，WorkflowToken 标题与 Workflow 标题相同。
字符串	workflowId	此 WorkflowToken 对象是其运行实例的工作流的标识符。
字符串	currentItemName	调用 getWorkflowTokenForId 时，工作流中正在运行的步骤的名称。
字符串	currentItemState	工作流中当前步骤的状态，具有以下可能值： ■ running：该步骤正在运行 ■ waiting：该步骤正在等待运行时参数，这些参数可由 answerWorkflowInput 提供 ■ waiting-signal：该步骤正在等待来自插件的外部事件 ■ canceled：该步骤已被用户或 API 集成的程序取消 ■ completed：该步骤已完成 ■ failed：该步骤遇到错误 每次更新此值时，都必须运行 getWorkflowTokenForId。 **注意** VMware 建议不要使用 currentItemState。globalState 属性使 currentItemState 成为多余的属性。
字符串	globalState	工作流的整体状态，具有以下可能值： ■ running：工作流正在运行 ■ waiting：工作流正在等待运行时参数，这些参数可由 answerWorkflowInput 提供 ■ waiting-signal：工作流正在等待外部事件 ■ canceled：工作流已被用户或应用程序取消 ■ completed：工作流已完成 ■ failed：工作流遇到错误 globalState 是工作流的整体状态。 每次更新此值时，都必须运行 getWorkflowTokenForId。
字符串	startDate	此工作流令牌的启动日期和时间 startDate 值在工作流启动时设置。获取令牌时，其 startDate 已初始化。

类型	值	描述
字符串	endDate	此工作流令牌结束的日期和时间（如果工作流令牌已完成）。 endDate 值在工作流运行结束时填充。 仅当工作流以 completed、failed 或 canceled 状态之一结束时，才会设置 endDate。
字符串	xmlContent	定义输入参数、输出参数、属性和错误消息的内容。属性和参数的值在 CDATA 元素中设置，而错误消息则在 <exception> 标记中设置，如以下示例所示。 <token> <atts> <stack> <att n='attstr' t='string' e='n'> <![CDATA[attribute]]>Attribute value</att> <att n='instr' t='string' e='n'> <![CDATA[]]>Input parameter value</att> <att n='outstr' t='string' e='n'> <![CDATA[]]>Output parameter value</att> </stack> </atts> <exception encoded='n'>Error message</exception> </token>

WorkflowTokenAttribute 对象

WorkflowTokenAttribute 对象表示一个正在运行的工作流实例的输入或输出参数。

WorkflowTokenAttribute 是 WorkflowToken 在开始时（某些情况下，是在运行时）传递到预定义 WorkflowParameter 的值。运行工作流时，将以 WorkflowTokenAttribute 对象的形式为该特定工作流提供输入参数。调用 executeWorkflow 操作时，该操作将采用一个 WorkflowTokenAttribute 对象数组作为参数，如以下示例所示。

```
public WorkflowToken executeWorkflow(String workflowId, String username,
String password, WorkflowTokenAttribute[] attributes);
```

工作流也使用 WorkflowTokenAttribute 作为运行工作流的输出参数。WorkflowTokenAttribute 包含通过运行 executeWorkflow 创建的已完成 WorkflowToken 的结果。通过调用 getWorkflowTokenResult，可以采用 WorkflowTokenAttribute 的形式收集 WorkflowToken 的结果，如以下示例所示。

```
public WorkflowTokenAttribute[] getWorkflowTokenResult(String workflowTokenId,
String username, String password);
```

此外，还可以将 WorkflowTokenAttribute 对象数组传递到 answerWorkflowInput 操作，以在工作流令牌运行时提供所需的输入参数。

```
public void answerWorkflowInput(String workflowTokenId,
WorkflowTokenAttribute[] answerInputs, String username, String password);
```

下表显示 WorkflowTokenAttribute 对象的属性。

类型	值	描述
字符串	name	输入或输出参数的名称
字符串	type	输入或输出参数的类型
字符串	value	value 属性表示此特定工作流令牌的输入或输出参数值（采用字符串形式）。 如果 type 为对象数组，则 value 是以下格式的字符串： "#{#<type1>#<value1>#;#<type2>#<value2>#...}#" 如果 value 属性指定了一个从插件获取的对象，则输入或输出参数值是指向相关对象的 dunesUri 字符串。以下示例显示 dunesUri 的格式。 dunes://service.dunes.ch/CustomSDKObject?id='*<对象_ID>*'&dunesName='*<插件_名称>*:*<对象_类型>*'

Web 服务 API 操作

Orchestrator Web 服务 API 提供一个作为 WSDL 操作使用的方法集合。

注意 除 echo、echoWorkflow 和 sendCustomEvent 之外，每个 Web 服务操作均使用 Orchestrator 用户名和密码来验证会话。如果使用了错误的用户名或密码，操作将引发异常。

answerWorkflowInput 操作

answerWorkflowInput 操作可在工作流运行时将信息从用户或外部应用程序传递到该工作流。

如果一个正在运行的工作流到达某个需要由用户操作或外部应用程序输入的阶段，WorkflowToken 将进入 waiting 状态，直到它收到来自 answerWorkflowInput 的输入。answerWorkflowInput 操作以 WorkflowTokenAttribute 对象数组的形式提供输入。

answerWorkflowInput 操作声明如以下示例所示。

```
public void answerWorkflowInput(String workflowTokenId, WorkflowTokenAttribute[] answerInputs,
String username, String password);
```

Web 服务仅对为运行工作流而提供的输入属性执行简单验证。Web 服务仅验证在 WorkflowTokenAttribute 对象中设置的属性是否为所需类型。Web 服务不会执行复杂验证，即不会验证您是否正确设置了所有 WorkflowTokenAttribute 对象的属性。Web 服务不访问工作流开发人员在工作流呈现方式中设置的参数属性。如果其中一个 WorkflowTokenAttribute 对象的属性未设置，或如果某个属性值不是工作流所期望的值，则 Web 服务将随无效的 WorkflowTokenAttribute 对象发送 answerWorkflowInput 请求。如果 WorkflowTokenAttribute 对象无效，工作流将失败，并在不通知 Web 服务应用程序的情况下进入 failed 状态。Web 服务应用程序可通过在工作流运行期间和之后调用 getWorkflowTokenStatus 操作，检查工作流是正常运行，还是运行失败。

类型	值	描述
字符串	workflowTokenId	正在等待由用户交互或外部应用程序输入的运行中工作流的 ID
WorkflowTokenAttribute 对象的数组	answerInputs	用户交互或外部应用程序的结果，以输入的形式向正在等待的工作流传递
字符串	username	Orchestrator 用户名
字符串	password	Orchestrator 密码

返回值

无返回值。如果向其传递的参数无效，则会引发异常。

cancelWorkflow 操作

`cancelWorkflow` 操作可取消工作流。

`cancelWorkflow` 操作的行为取决于它所取消的工作流。已取消的工作流将在 Orchestrator 服务器中停止运行，并进入 `canceled` 状态，但是已运行或已启动运行的相关操作将不会停止或反向运行。例如，当取消某个工作流时，如果该工作流正在执行启动虚拟机的操作，虚拟机将不会停止启动，而如果该虚拟机已经启动，则不会关闭电源。

`cancelWorkflow` 操作声明如下。

```
public void cancelWorkflow(String workflowTokenId, String username, String password);
```

类型	值	描述
字符串	`workflowTokenId`	要取消的运行中工作流的标识符
字符串	`username`	Orchestrator 用户名
字符串	`password`	Orchestrator 密码

返回值

无返回值。如果向其传递的参数无效，`cancelWorkflow` 操作将返回异常。

echoWorkflow 操作

`echoWorkflow` 操作可通过检查序列化来测试与 Web 服务的连接。

如果正在连接到版本较早的 Web 服务实施，`echoWorkflow` 操作将会提供一个有用的调试工具。调用此操作可验证到服务器的连接，方法是检查序列化和反序列化操作是否正常工作。

`echoWorkflow` 操作声明如下。

```
public Workflow echoWorkflow(Workflow workflow);
```

类型	值	描述
工作流	`workflow`	`echoWorkflow` 操作将 `Workflow` 对象作为参数。如果连接和序列化都正常工作，将返回相同的工作流。

返回值

返回与作为输入参数提供的对象相同的 `Workflow` 对象。

executeWorkflow 操作

`executeWorkflow` 操作将运行指定的工作流。

`executeWorkflow` 将 `WorkflowTokenAttribute` 对象数组作为输入参数，这些参数将为此特定工作流实例提供运行所需的特定属性。

`executeWorkflow` 操作声明如下。

```
public WorkflowToken executeWorkflow(String workflowId, String username, String password,
WorkflowTokenAttribute[] attributes);
```

类型	值	描述
字符串	`workflowId`	要运行的工作流的标识符
字符串	`username`	Orchestrator 用户名

类型	值	描述
字符串	password	Orchestrator 密码
WorkflowTokenAttribute 实例的数组	workflowInputs	运行工作流所需的输入参数数组

返回值

返回一个 WorkflowToken 对象。如果向其传递的参数无效，则返回异常。

find 操作

find 操作可查找与某个特定查询相对应的元素。

find 操作可通过搜索某个特定名称，获取任何类型的对象。查询结果以 QueryResult 对象的形式提供，此对象包含一个 FinderResult 对象数组以及一个总数计数器。查询本身将作为第二个参数传递到 find，如以下操作声明所示。

```
public QueryResult find(String type, String query, String username, String password);
```

查询分析由包含要查找的对象的插件执行。find 操作所使用的查询语言由相关插件定义。因此，query 参数的语法会有所不同，具体取决于插件的实施。大多数正式受支持的 Orchestrator 插件不在清单中存储任何对象，因此它们不会公开任何可搜索到的对象。表 7‑1 提供了正式受支持的 Orchestrator 插件的查询参数的语法和行为。

该表介绍了每个受支持的 Orchestrator 插件的 find 操作的 query 参数语法。

表 7‑1 Orchestrator 插件的查询语法

Orchestrator 插件	查询参数语法	查询行为
数据库（如 Lifecycle Manager）	字符串	在 SQL 数据库表中搜索对象名称，Orchestrator 在 SQL WHERE 关键字搜索中设置搜索字符串。它先在数据库中搜索主键，然后再搜索对象 ID。
枚举	不适用	枚举插件不在清单中存储任何内容。可以对包含枚举类型的每个数据类型查找枚举。
Jakarta 通用工具包	不适用	Jakarta 插件不在清单中存储任何内容。
JDBC	不适用	JDBC 插件不在清单中存储任何内容。
库	不适用	库插件不在清单中存储任何内容。
邮件	不适用	邮件插件不在清单中存储任何内容。
SSH	如果已将 Orchestrator 配置为使用 SSH 连接，则可以查询 SSH 命令。	SSH 插件不在清单中存储任何内容。
VMware Infrastructure 3.5	字符串或 Null	忽略查询字符串，并返回指定类型的所有对象。
vCenter Server 4.0	字符串或 Null	忽略查询字符串，并返回指定类型的所有对象。
XML	不适用	XML 插件不在清单中存储任何内容。

开发插件时，可以将查询语言定义为使用 find 通过自定义插件搜索已命名的对象。此定义不是强制性的。查询参数的语法完全取决于插件实施的查询语言。要避免定义查询语言，请使 find 返回所有对象，如 VMware Infrastructure 插件那样。

QueryResult 返回的对象数组的大小取决于用于查询的插件的定义。对于使用标准 Orchestrator 插件进行的查询，数组将包含无限数量的 FinderResult 对象。但是，第三方插件的开发人员可以对查询返回的结果限制数量限制。在此类情况下，如果 totalCount 的值大于 FinderResult 对象数组中的对象数，数组将不会包含查询清单中找到的所有对象。但 totalCount 属性会报告找到的 FinderResult 对象的总数。totalCount 属性可以为负值，这表示插件无法确定该插件中有多少相应对象。

类型	值	描述
字符串	type	所查找对象的类型。
字符串	query	查询。 该查询是包含在引号中的字符串。任何属于 type 参数指定的类型的对象，其名称都与 QueryResult 中返回的查询字符串相匹配。
字符串	username	Orchestrator 用户名。
字符串	password	Orchestrator 密码。

返回值

以 QueryResult 对象的形式返回查询结果。

如果 find 找不到匹配的对象，QueryResult.getTotalCount 将返回 0，而 QueryResult.getElement 将返回 Null。

如果服务器无法识别所搜索的对象类型或插件，find 将引发异常。如果向其传递的参数无效，则 find 也会返回异常。

findForId 操作

findForId 操作可根据某个特定 FinderResult 对象的 type 和 id 属性来搜索该 FinderResult 对象。

可以使用 findForId 操作获取通过使用其他 find* 操作找到的 FinderResult 对象的相关信息。例如，可以使用 findForId 方法获取通过 find 操作找到的 FinderResult 对象的状态。

findForId 操作声明如以下示例所示。

```
public FinderResult findForId(String type, String id, String username,
String password);
```

类型	值	描述
字符串	type	所查找对象的类型。
字符串	id	所查找对象的 ID。
字符串	username	Orchestrator 用户名。
字符串	password	Orchestrator 密码。

返回值

返回 FinderResult 对象，该对象包含找到的对象的详细信息。如果向其传递的参数无效，则返回 Null。

findRelation 操作

findRelation 操作可在清单中查找属于某一特定父元素或某一父元素类型的所有子元素。

如果要开发树查看器以查看库中的对象，则需要了解子对象与其父对象的相关方式。findRelation 操作声明如下。

```
public FinderResult[] findRelation(String parentType, String parentId,
String relation, String username, String password);
```

类型	值	描述
字符串	parentType	父对象的类型。 parentType 属性可以是插件的名称，也可用于指定在更小范围内定义的父对象。例如，您可以将 parentType 指定为 "VC:"以获取位于 VMware vCenter Server 4.0 插件的根位置处的对象，或者也可指定某一特定文件夹，如 "VC:VmFolder"。
字符串	parentId	特定父对象的 ID。 parentId 参数可用于查找特定父对象的子对象（如果知道其 ID）。
字符串	relation	关系的名称。 调用 findRelation 将返回由 parentId 标识的某个父元素下的所有子元素。如果忽略 parentId（parentType 不是清单的根类型），则 findRelation 操作将返回 Null。 有关详细信息，请参见表 7-2。
字符串	username	Orchestrator 用户名。
字符串	password	Orchestrator 密码。

关系类型

relation 属性类型由插件定义。关系的有效性取决于父类型。

此表列出了由 Orchestrator 提供的每个标准插件定义的关系类型。

表 7-2 标准 Orchestrator 关系类型

插件	关系名称	关系类型
枚举	无关系	无关系
Jakarta Commons Net	无关系	无关系
JDBC	无关系	无关系
库	无关系	无关系
邮件	无关系	无关系
网络连接		■ IpAddress ■ IPV4Address ■ MacAddressPool ■ NetworkDomain ■ Proxy ■ Subnet ■ Range
SSH		■ File ■ Folder ■ RootFolder ■ SshConnection

表 7‑2 标准 Orchestrator 关系类型 （续）

插件	关系名称	关系类型
vCenter Server	■ getComputeResource_ClusterComputeResource() ■ getComputeResource_ComputeResource() ■ getDatacenter() ■ getDatastore() ■ getDatastoreFolder() ■ getFolder() ■ getFolder() ■ getFolder() ■ getFolder() ■ getFolder() ■ getHost() ■ getHostFolder() ■ getNetwork() ■ getNetworkFolder() ■ getNetwork_DistributedVirtualPortgroup() ■ getNetwork_Network() ■ getOwner() ■ getParentFolder() ■ getPortgroup() ■ getRecentTask() ■ getResourcePool() ■ getResourcePool_ResourcePool() ■ getResourcePool_VirtualApp() ■ getRootFolder() ■ getSdkConnections() ■ getVm() ■ getVmFolder() ■ getVmSnapshot()	■ ComputeResource ■ ComputeResource ■ Datacenter ■ Datastore ■ DatastoreFolder ■ DatacenterFolder ■ DatastoreFolder ■ HostFolder ■ NetworkFolder ■ VmFolder ■ HostSystem ■ HostFolder ■ Network ■ NetworkFolder ■ DistributedVirtualPortgroup ■ Network ■ ComputeResource ■ VmFolder ■ DistributedVirtualPortgroup ■ Task ■ ResourcePool ■ ResourcePool ■ VirtualApp ■ DatacenterFolder ■ SdkConnection ■ VirtualMachine ■ VmFolder ■ VirtualMachineSnapshot
XML	无关系	无关系

`relation` 属性也可以引用在每个插件的 `vso.xml` 文件中指定的关系类型。以下示例是来自网络连接插件 `vso.xml` 文件的节选内容。

```
[...]
<relations>
                <relation name="Subnet" type="Class:Subnet"/>
                <relation name="Range" type="Class:Range"/>
                <relation name="NetworkDomain" type="Class:NetworkDomain"/>
                <relation name="MacAddressPool" type="Class:MacAddressPool"/>
    </relations>
[...]
```

除了表 7‑2 中列出的关系类型之外，Orchestrator 还定义了 `CHILDREN` 关系，用于表示所有关系类型。

返回值

返回一个 `FinderResult` 对象的列表。

如果未找到子元素，或者如果向其传递的参数无效，将返回异常。

getAllPlugin 操作

getAllPlugin 操作可返回安装在 Orchestrator 中的所有插件的描述。

重要事项 getAllPlugin 操作已弃用。请改用 getAllPlugins。

---

getAllPlugins 操作

getAllPlugins 操作可返回安装在 Orchestrator 中的所有插件的描述。

使用 Orchestrator 执行的很多操作都取决于通过插件启用的函数。工作流可能取决于是否存在某些自定义插件，或者取决于管理员已禁用的标准插件。因此，在运行工作流之前，可以检查是否存在必要的插件。如果没有必要的插件，则工作流使用的某些对象类型可能不存在。

getAllPlugins 操作以 ModuleInfo 对象数组的形式列出了所有可用插件。ModuleInfo 对象用于存储每个插件的名称、版本、描述和名称属性。Web 服务应用程序可根据是否存在某些插件模块或版本，使用这些属性来修改自身的行为。

getAllPlugins 操作声明如下。

```
public ModuleInfo[] getAllPlugins(username, password);
```

注意 getAllPlugins 操作可替代弃用的操作 getAllPlugin。

下表介绍 getAllPlugins 操作的属性。

类型	值	描述
字符串	username	Orchestrator 用户名。
字符串	password	Orchestrator 密码。

返回值

以 ModuleInfo 对象的形式返回一个插件描述列表。

getAllWorkflows 操作

getAllWorkflows 操作可用于查找所有可用的工作流。

getAllWorkflows 操作以 Workflow 对象数组的形式列出 Orchestrator 服务器中的所有可用工作流。对于必须列出工作流相关信息（如工作流的名称、ID 等）的程序，getAllWorkflows 操作也非常有用。Workflow 对象显示有关工作流的所有相关信息。

getAllWorkflows 操作声明如下。

```
public Workflow[] getAllWorkflows(String username, String password);
```

类型	值	描述
字符串	username	Orchestrator 用户名。
字符串	password	Orchestrator 密码。

返回值

返回 Workflow 对象数组。

getWorkflowForId 操作

getWorkflowForId 操作可检索由其唯一 ID 标识的工作流。

如果知道某个特定工作流的 ID，则可以使用 getWorkflowForID 操作来获取该工作流对象。通过不同插件运行的多个工作流可能拥有相同的名称。获取工作流最安全的方式是使用 getWorkflowsWithName 操作获取其 ID，而不是按名称来获取它们。

通过检验工作流的 workflowID 属性，可以找到工作流 ID，如以下示例所示。

```
String workflowId = workflows[0].getId();
```

getWorkflowForId 操作声明如下。

```
public Workflow getWorkflowForId(String workflowId, String username,
String password);
```

类型	值	描述
字符串	workflowId	要检索的工作流的 ID。
字符串	username	Orchestrator 用户名。
字符串	password	Orchestrator 密码。

返回值

返回对应于所提供 ID 的 Workflow 对象。如果向其传递的参数无效，则返回 Null。

getWorkflowsWithName 操作

getWorkflowsWithName 操作按其名称搜索工作流。

getWorkflowsWithName 操作声明如下。

```
public Workflow[] getWorkflowsWithName(String workflowName, String username, String password);
```

如果知道某个特定工作流的名称（或名称的一部分），则可通过调用 getWorkflowsWithName 来获取此工作流。getWorkflowsWithName 操作可返回一个工作流数组，因此可用于同时查找多个工作流。

Orchestrator 4.0 提供两个 VMware vCenter Server 插件，一个用于 vCenter Server 3.x，另一个用于 vCenter Server 4.0。如果仅按名称获取工作流，则不一定能够轻松地确定这些工作流是通过 vCenter Server 3.x 插件运行，还是通过 vCenter Server 4.0 插件运行。这两个插件可能同时存在名称相同的工作流，除非工作流开发人员为工作流名称附加了插件标识符，如 VI3、VI3.5 或 VC。即使工作流开发人员添加了建议的名称后缀，获取工作流最安全的方式也还是使用 getWorkflowForId 操作按其 ID 来获取。但是，如果使用 getWorkflowsWithName 操作，则可以检查工作流的输入和输出参数，以明确工作流在哪个插件中运行。

类型	值	描述
字符串	workflowName	要查找的工作流的名称。 workflowName 属性的值可以是完整名称，也可以是通配符 (*)，通配符将向用户返回所有可用的工作流。您也可以搜索部分名称。例如，如果输入 ***Clone** 或 **Clone*** 作为 workflowName，则将返回所有名称中包含单词 Clone 的工作流。
字符串	username	Orchestrator 用户名。
字符串	password	Orchestrator 密码。

返回值

返回对应于所提供名称或名称片段的 Workflow 对象数组。即使只找到一个工作流，工作流也会以数组的形式返回。如果向其传递的参数无效，则返回 Null。

getWorkflowTokenForId 操作

getWorkflowTokenForId 操作可查找某个特定工作流令牌的 WorkflowToken 对象。

getWorkflowTokenForId 操作声明如下。

```
public WorkflowToken getWorkflowTokenForId(String workflowTokenId, String username,
String password);
```

单个线程或函数可以运行多个工作流。getWorkflowTokenForId 操作允许中心过程或线程跟踪每个工作流的进度。使用getWorkflowTokenForId可以访问有关某个特定WorkflowToken的所有信息，因为，尽管检查令牌状态仅需要 ID，但通常获取某个给定令牌的所有相关信息会非常有用。

类型	值	描述
字符串	workflowTokenId	此工作流运行的 ID
字符串	username	Orchestrator 用户名。
字符串	password	Orchestrator 密码。

返回值

返回与所提供的工作流令牌 ID 对应的某个特定工作流令牌的 WorkflowToken 对象。

getWorkflowTokenResult 操作

getWorkflowTokenResult 操作可获取某个给定工作流的运行结果。

通过调用 getWorkflowTokenResult，可以查看由 WorkflowToken 对象生成的结果。工作流的运行结果将以 WorkflowTokenAttribute 对象数组的形式传递，这些对象包含工作流在运行期间设置的输出参数。WorkflowTokenAttribute 输出对象的结构与在工作流启动时传递到该工作流的输入参数的结构相同。这些参数拥有名称、类型和值。

您可以在工作流完成之前获取结果。如果工作流已经设置了输出参数，则在工作流运行时可通过调用 getWorkflowTokenResult 来获取这些参数的值。此方法使工作流可以在仍处于 running 状态时将其结果传递到外部系统中。您也可以使用 getWorkflowTokenResult 获取处于 failed、waiting 和 canceled 状态的工作流的结果，以显示该工作流进入非运行或未完成状态之前的结果。

Any 类型的对象未正确进行反序列化。因此，如果某个工作流令牌的其中一个属性为 Any 类型，则无法对该令牌调用 getWorkflowTokenResult。如果指定了正确的对象类型（例如 VC:VirtualMachine），则 getWorkflowTokenResult 将返回正确的 dunesURI 值。

如果 getWorkflowTokenResult 获取的对象是普通 Java 对象，则可通过使用标准 Java API 对其执行反序列化，但是要执行此操作，您必须在类路径中包括相关的Java类。例如，如果获取的对象为VirtualMachineRuntimeInfo 类型，则必须在类路径中包括 VirtualMachineRuntimeInfo.class 或 vmware-vmosdk-vc40.jar。在*安装-目录* \VMware\Orchestrator\app-server\server\vmo\tmp\dars\vmware-vmosdk-vc40.dar\lib 中查找 vmware-vmosdk-vc40.jar 文件。

getWorkflowTokenResult 操作声明如下。

```
public WorkflowTokenAttribute[] getWorkflowTokenResult(String workflowTokenId,
String username, String password);
```

类型	值	描述
字符串	workflowTokenId	此特定工作流运行的 ID
字符串	username	Orchestrator 用户名。
字符串	password	Orchestrator 密码。

返回值

返回对应于所提供工作流令牌 ID 的 WorkflowTokenAttribute 对象数组。如果向其传递的参数无效，则返回 Null。

getWorkflowTokenStatus 操作

getWorkflowTokenStatus 操作可获取特定工作流令牌的 globalStatus。

getWorkFlowTokenStatus 操作可在工作流或工作流数组运行时检查其状态。getWorkFlowTokenStatus 操作可从正在运行的 WorkflowToken 对象（由其 workflowTokenId 标识）中获取 globalStatus 值。globalStatus 值可以是以下之一。

- running：工作流正在运行
- waiting：工作流正在等待运行时参数，这些参数可由 answerWorkflowInput 提供
- waiting-signal：工作流正在等待外部事件
- canceled：工作流已被用户或应用程序取消
- completed：工作流已完成
- failed：工作流遇到错误

getWorkflowTokenStatus 操作声明如下。

```
public String[] getWorkflowTokenStatus(String[] workflowTokenID, String username,
String password);
```

类型	值	描述
字符串数组	workflowTokenId	工作流令牌 ID 的列表。
字符串	username	Orchestrator 用户名。
字符串	password	Orchestrator 密码。

返回值

返回一个工作流令牌状态值的列表。返回的值是每个工作流令牌的 globalStatus 的字符串数组，按其 workflowTokenID 值排序。如果向其传递的参数无效，则返回 Null。

相关信息

有关相关信息，请参见第 165 页，“WorkflowToken 对象”。

hasChildrenInRelation 操作

hasChildrenInRelation 操作可检查某个给定关系类型是否拥有任何子对象。

在某些情况下，通过对象与其他对象之间的关系，可以很方便地找到所需的对象。您可以通过对另一个对象调用 findRelation 操作，按给定关系获取与该对象相关的所有对象。findRelation 操作仅可用于查找已知对象的关系。hasChildrenInRelation 操作可检查是否存在具有给定 relation 属性的对象。hasChildrenInRelation 可检查是否存在按给定关系类型与其父对象相关的其他对象的子对象。例如，虚拟机的快照是原始虚拟机的子对象。您可以通过检查属于其他虚拟机的子对象的所有虚拟机来找到所有快照。

如果要开发树查看器以查看库中的对象，则需要了解子对象与其父对象的相关方式。hasChildrenInRelation 操作声明如下。

```
public int hasChildrenInRelation(String parentType, String parentId, String relation, String
username, String password);
```

类型	值	描述
字符串	parentType	父对象的类型。可通过指定父对象类型来缩小搜索范围，这样便可将结果限制为按给定关系与给定类型的父对象相关的子对象。 此值可以为 Null，在此情况下，hasChildrenInRelation 将查找按指定关系类型与所有父对象类型相关的子对象。
字符串	parentId	特定父对象的 ID。 您可以通过指定 parentId 来检查按给定关系与特定父对象相关的子对象。如果特定父对象拥有大量按不同关系类型与其相关的子对象，则此检查非常有用。findRelation 操作可返回该父对象的所有子对象，而不管是什么关系类型。hasChildrenInRelation 仅检查是否存在按所需关系类型相关的子对象。 如果对对象层次结构的根对象调用 hasChildrenInRelation，则此值可能为 Null。
字符串	relation	子对象与其父对象相关所遵循的关系类型。 关系类型在每个插件的 vso.xml 文件中指定。
字符串	username	Orchestrator 用户名。
字符串	password	Orchestrator 密码。

返回值

返回以下值之一：

1	是，存在指定关系类型的子对象
-1	否，不存在指定关系类型的子对象
0	未知，或输入参数无效

相关信息

有关详细信息，请参见第 171 页，“关系类型”。

hasRights 操作

hasRights 操作可检查用户是否拥有查看、编辑和运行工作流的权限。

要检查您对工作流中的某个项目拥有哪些权限，必须拥有查看此项目的权限。如果您只有编辑或运行某个项目的权限，则无法查看您对此项目拥有哪些权限，而且 hasRights 会返回 False。

在 Orchestrator 客户端的 **Authorizations** 窗格中设置工作流的用户权限。Web 服务应用程序可通过调用 hasRights 操作来检查这些权限。在以下示例中，hasRights 将检查用户是否拥有读取工作流的权限。

hasRights(*taskId*, *username*, *password*, 'r')

类型	值	描述
字符串	taskId	要检查用户权限的工作流的 ID。
字符串	username	Orchestrator 用户名。
字符串	password	Orchestrator 密码。
整型	rights	■ a：管理员可以更改对象的权限 ■ c：用户可以编辑工作流。 ■ I：用户可以检查工作流架构和脚本。 ■ r：用户可以查看工作流（但不能查看架构或脚本）。 ■ x：用户可以运行工作流。 **注意** 用户权限不具累积性。要执行工作流中的所有可能的任务，用户必须拥有所有权限。

返回值

将返回以下值：

- 如果用户拥有指定的工作流权限，将返回 True。
- 如果用户没有指定的工作流权限，将返回 False。

如果工作流不存在或如果调用 hasRights 的用户没有查看工作流的权限，则 hasRights 操作将返回 "Unable to find workflow" 异常。

sendCustomEvent 操作

sendCustomEvent 操作可将工作流与外部事件同步。

```
public void sendCustomEvent(String eventName, String serializedProperties);
```

sendCustomEvent 操作可将来自 Web 服务客户端的消息发送到正在等待特定事件发生再运行的工作流中。处于等待状态的工作流会在收到来自 sendCustomEvent 的消息后恢复其运行。

调用 sendCustomEvent 以在事件发生时发送消息的自定义事件，可以是 Orchestrator 可运行的任何脚本、工作流或操作。例如，一个工作流可能会在运行时使用 sendCustomEvent 来触发另一个工作流，该工作流将在发送消息的工作流在运行过程中执行特定操作时重新加载所有 Orchestrator 插件。

sendCustomEvent 发送的消息是简单触发器，其格式不向用户公开。服务器一收到消息，该消息便立即触发处于等待状态的工作流，使其运行。

重要事项 对 sendCustomEvent 操作的访问不受用户名和密码组合的保护。因此，VMware 建议仅在安全的内部部署中使用此功能。例如，不要在通过 Internet 公开运行的部署中使用此操作。

类型	值	描述
字符串	eventName	eventName 属性是工作流在运行之前等待的事件的名称。传递到 sendCustomEvent 的 eventName 字符串必须与 Event 对象的名称匹配，该名称在定义自定义事件的脚本、操作或工作流中声明。
字符串	serializedProperties	serializedProperties 属性可将要传递到处于等待状态的工作流的参数定义为一系列名称/值对。serializedProperties 的语法如下： "*name1*=*value1*\n*name2*=*value2*\n*name3*=*value3*" 如果工作流不需要输入参数，则 serializedProperties 属性可以为 Null，也可以省略。

返回值

如果无返回值，将通知应用程序 sendCustomEvent 操作运行成功。

如果向其传递的参数无效，则 sendCustomEvent 操作将返回异常。

接收来自 sendCustomEvent 的消息

等待来自 sendCustomEvent 的消息再运行的工作流必须声明其正在等待的事件，方法是从 Orchestrator API 调用 System.waitCustomEvent 或 System.waitCustomEventUntil 操作。以下示例显示对 waitCustomEvent 的两个调用。

```
System.waitCustomEvent("internal", customEventKey, myDate);
System.waitCustomEvent("external", customEventKey, myDate);
```

waitForCustomEvent 操作的参数如下所示。

internal / external	所等待的事件来自另一个工作流 (internal) 或来自 Web 服务应用程序 (external)。
customEventKey	所等待的事件的名称。
myDate	waitForCustomEventUntil 等待来自 sendCustomEvent 的消息的截止日期。

simpleExecuteWorkflow 操作

simpleExecuteWorkflow 操作使用字符串属性启动工作流。

重要事项 此操作已弃用。请不要使用 simpleExecuteWorkflow。

类型	值	描述
字符串	workflowId	要运行的工作流的 ID。
字符串	username	Orchestrator 用户名。
字符串	password	Orchestrator 密码。
字符串	attributes	attributes 参数的格式是一个由逗号分隔的属性列表。因为逗号当作分隔符使用，所以包含逗号的属性名称字符串无法正确处理。 每个属性均由其名称、类型和值表示，如以下示例所示。 *Name1*,*Type1*,*Value1*,*Name2*,*Type2*,*Value2*

返回值

运行工作流。如果向其传递的参数无效，则返回异常。

开发 Web 视图 8

Orchestrator 提供一个 Web 2.0 前端，该前端包含一组可用于创建基于浏览器的用户前端的 Web 组件。

本章讨论了以下主题：

Web 视图概述

Orchestrator 提供一个由可自定义组件组成的库，以便您可以使用 Web 2.0 前端或 Web 视图访问已耦合的对象。

Web 视图是由多个网页、样式表、图标和横幅组成的软件包，可用于表示一个完整的网站。Web 视图可以包含一些特殊组件，通过这些组件可从基于浏览器的客户端访问所有 Orchestrator 功能。网页通常通过 Web 设计工具在外部创建，然后再导入到 Web 视图软件包中。通过将完整的网页导入到 Web 视图软件包中，可将 Web 设计过程与 Orchestrator Web 视图组件的开发过程分离。

Orchestrator Web 视图使用 Ajax 技术，因此您不必重新加载完整的网页即可动态更新内容。Orchestrator 在服务器端提供一个 Tapestry 组件库，并在客户端提供一个 Dojo 组件库，以帮助生成自定义的 Web 视图。

启动 Web 视图

从 Orchestrator 客户端中的“Web Views”视图启动 Web 视图。

前提条件

必须将包含 Web 视图的软件包导入到 Orchestrator 中。Orchestrator 提供一个名为 `weboperator` 的 Web 视图，可将其用于启动工作流，此外还可以进行扩展或自定义。

步骤

1 在 Orchestrator 客户端选项卡中单击“Web Views”。

  此时将显示 `weboperator` Web 视图以及其他已导入到 Orchestrator 中的 Web 视图。

2 右键单击 `weboperator`，然后选择 **Publish**。

3 打开浏览器，并转到 `http://`*设备_名称_或_IP_地址*`:8280/`。

  Orchestrator 主页打开，其中显示指向 Web 视图列表和 Orchestrator 配置的链接。

4 单击 **Web View List**。

5 单击 **weboperator**。

6 使用 Orchestrator 用户名和密码登录。

此时将显示一个 Web 界面，可用于访问 Orchestrator 库中的工作流。可以使用此 Web 视图通过网络与工作流交互。

您已经启动了一个 Web 视图。

编辑 Web 视图

可以在 Orchestrator 客户端的 **Web Views** 视图中编辑现有 Web 视图。

步骤

1 在 Orchestrator 客户端中单击 **Web Views** 视图。

2 如果 Web 视图正在运行，右键单击 Web 视图，然后选择 **Unpublish**。

3 右键单击 Web 视图并选择 **Edit**。

此时将打开 Web 视图编辑器，在其中可以编辑常规设置、添加或删除元素、定义 Web 视图属性以及跟踪影响 Web 视图的事件。

Web 视图的文件结构

开发 Web 视图时，必须在一个工作文件夹中保存组成 Web 视图的网页和 Web 视图组件的集合。Web 视图工作文件夹必须符合基本文件命名和文件结构规则。

可以将在其中开发 Web 视图页和组件的工作文件夹命名为任何适当的名称。但是，此工作文件夹必须在其根目录包含以下文件和文件夹。

_<WebView_Folder>_/default.html	Web 视图的主页。所有 Web 视图都必须在工作文件夹的根目录中包含 `default.html` 文件。
_<WebView_Folder>_/components/	包含 Web 视图组件的 JWC 文件及其关联的 HTML 模板。可以在 `components` 文件夹中创建子文件夹，但 `components` 文件夹自身必须位于工作文件夹的根目录下。

`default.html` 文件和 `components` 文件夹是 Web 视图必须包含的唯一强制元素。可以在 Web 视图文件夹中添加其他文件和文件夹，并用任何方式组织整理这些文件和文件夹。您可以在 Web 视图文件夹中的任何地方包含非 Web 视图组件模板的 HTML 文件。

将 Web 视图组件添加到 HTML 页

您可以通过将 Tapestry 组件添加到 HTML 页构建 Orchestrator Web 视图。Orchestrator 提供一个自定义 Orchestrator Tapestry 组件库供您使用。此外，您还可以使用来自 Web 视图标配的 Tapestry 框架 4.0 的各个组件。

您通过将 `jwcid` 属性添加到网页的 HTML 标记中来添加 Tapestry 组件。`jwcid` 属性是将 HTML 组件映射到 Tapestry 组件的 Hibernate 属性，可用于将 Tapestry 组件函数添加到网页中。

有关 Tapestry 框架的详细信息，请参见 http://tapestry.apache.org/tapestry4/。

有关 Hibernate 的详细信息，请参见 http://www.hibernate.org/。

将 jwcid 属性添加到网页中

通过将 `jwcid` 属性添加到 HTML 页中，可以将 Orchestrator 和 Tapestry 组件添加到 Web 视图的网页中。

前提条件

您必须具有一个作为 Orchestrator Web 视图的 Web 前端实施的 HTML 页。

步骤

1 在 HTML 编辑器中打开作为 Orchestrator Web 视图实施的 HTML 页。

2 将 `jwcid` 属性添加到用于启动 Orchestrator 或 Tapestry 组件的 HTML 标记中。

   例如，在 Web 视图中单击链接时，以下 `jwcid` 属性示例会实例化用于启动指定工作流的 Orchestrator `WorkflowLink` 组件。

   ```
   <a jwcid="@WorkflowLink" workflow="startVmWorkflow">Start a Virtual Machine</a>
   ```

添加 `jwcid` 属性以在网页中实例化 Orchestrator 或 Tapestry Web 视图组件。

初始化 Tapestry 组件

您通过在 Web 视图页的 HTML 标记中设置 `jwcid` 属性初始化 Tapestry 组件。

要将 `jwcid` 属性指向定义某个特定操作的 Java 类，定义组件行为的 Tapestry 组件 Java 类的名称应以 `@` 字符为前缀，如下例所示。

```
<div jwcid="@Border">
```

在 `@` 字符前可以放置一个唯一标识符，如下例所示。

```
jwcid="myBorderComponent@Border"
```

以上示例中的唯一标识符使您可以在整个 HTML 页中重用 `Border` 类，这可通过引用 `myBorderComponent` 实现。

注意 如果要在 `components` 文件夹中创建子文件夹，必须指定组件的完整路径。例如，如果要包括 *WebView_Folder*`/components/layout/` 子文件夹，必须如下例所示设置 `jwcid` 属性：

```
<vmo jwcid="@layout/Border">
```

通用 Web 视图组件参数

Web 视图组件可通过各种不同方式获取数据。您可以通过 Web 视图组件的参数定义其获取数据的方式。

以下组件参数可获取 Web 视图组件的数据。您在组件 HTML 模板的 `jwcid` 属性中声明参数。

`url` URL 指向组件所需的数据。

`action` `Action` 类型的 Web 视图属性，包含用于检索数据的操作。

`actionParameters` 要传递到用于检索数据的操作的参数数组。

`actionPageUrl` 操作从中检索数据的 URL。

`attribute` 包含组件所需数据的属性。

创建 Web 视图组件

您可以使用 Orchestrator Web 视图组件以及由 Tapestry 框架定义的组件。此外，您还可以创建自定义 Web 视图组件来执行来自网页的函数。

Tapestry Web 视图组件由规范文件、模板文件和 Java 类文件组成。Orchestrator 允许创建 Web 视图组件。但您无法创建 Tapestry 组件 Java 类。必须使用现有组件的 Java 类。

要创建组件，必须创建规范文件和模板文件。

Web 视图组件文件

Web 视图组件符合 Tapestry 框架标准。它必须包含组成 Tapestry 组件的标准文件。

Web 视图组件由以下文件组成。

Tapestry 组件模板	包含 Tapestry 组件布局的 HTML 文件。Tapestry 组件是添加到网页中的可重用对象。
Tapestry 组件规范	用于引用 Tapestry 组件 Java 类以及 Tapestry DTD 定义的扩展名为 `.jwc` 的文件。
Tapestry 组件 Java 类	一种指定 Tapestry 组件行为的 Java 类。

重要事项 HTML 组件模板文件和 Tapestry 组件规范的文件名必须相同。例如，如果组件模板命名为 `MyComponent.html`，那么必须将关联的组件规范命名为 `MyComponent.jwc`。

创建 Web 视图组件

要创建组件，必须创建一个组件规范文件和一个组件模板文件。这些文件用于定义组件的行为以及该组件如何出现在 HTML 页中。

前提条件

您必须能够访问 Orchestrator Web 视图 API 的 Java 类。

步骤

1 在 Web 视图文件夹的根目录下创建一个名为 `components` 的文件夹。

2 在 `components` 文件夹中，创建一个扩展名为 `.jwc` 的组件规范文件。

此文件引用一个现有 Java 类，用于为此 Web 视图组件提供函数。

例如，以下示例文件 `MyAbout.jwc` 通过 Orchestrator Web 服务 API 引用 Tapestry DTD 并指定 `About` 组件的 Java 类。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE component-specification
PUBLIC "-//Apache Software Foundation//Tapestry Specification 4.0//EN"
"http://jakarta.apache.org/tapestry/dtd/Tapestry_4_0.dtd">
<component-specification class="ch.dunes.web.webview.components.About"
allow-body="yes" allow-informal-parameters="yes"
</component-specification>
```

3 在 `components` 文件夹中，创建扩展名为 `.html` 的关联模板文件。

此文件是 Web 视图页的模板。

例如，用于定义 `MyAbout.jwc` 模板的 `MyAbout.html` 可以包含引用 Tapestry 组件 `content` 属性的 `jwcid` 属性。可以在 `content` 属性引用的文件中定义网页的内容。

```
<span jwcid="$content$">
<h5>
ABOUT US (NEW)
</h5>
</span>
```

您创建了一个 Web 视图组件。

创建 Web 视图

Orchestrator 提供一个 Web 视图模板，可以帮助您创建 Web 视图。最简单的 Web 视图创建方式就是使用此模板。

使用模板创建 Web 视图的过程包括将模板导入到 Orchestrator 中，以及修改所包含的文件以满足您的要求。此信息介绍如何修改此模板以创建名为 Virtual Machine Manager 的 Web 视图。

步骤

1 导入 Web 视图模板第 186 页，

Orchestrator 提供一个 Web 视图模板，可用于创建 Web 视图。

2 将 Web 视图导出到工作文件夹第 186 页，

要想简化 Web 视图的开发过程，请将工作流模板导出到可在其中处理它的本地文件夹中。

3 配置 Web 视图开发服务器第 187 页，

可以将 Orchestrator 服务器配置为从工作文件夹发布 Web 视图。此过程使您可以在开发 Web 视图时，不必将它导入 Orchestrator 服务器即可预览该视图。

4 编辑 Web 视图前端页第 188 页，

在 `default.html` 文件中创建 Web 视图的前端页。

5 创建用于获取虚拟机的操作第 189 页，

可以将 Virtual Machine Manager Web 视图的前端页自定义为在虚拟文件夹中显示所有虚拟机的列表。必须创建一个操作才能获取虚拟机。

6 设置 Web 视图授权第 190 页，

通过设置授权，可以定义哪些用户有权访问 Web 视图。

7 设置 Web 视图属性第 190 页，

您可以设置 Web 视图属性以将其指向要执行的操作。通过在 Orchestrator 客户端中编辑 Web 视图，可以设置 Web 视图的属性。

8 将组件添加到网页第 191 页，

可以将组件添加到执行耦合操作的 Web 视图页中。

9 自定义 Web 视图界面第 192 页，

Web 视图开发的最后一个阶段是自定义界面。

导入 Web 视图模板

Orchestrator 提供一个 Web 视图模板，可用于创建 Web 视图。

创建 Web 视图时，自定义现有模板比重新创建模板更容易。

前提条件

必须已将 Orchestrator 连接到正在运行的包含多台虚拟机的 vCenter Server 4.0 实例。

步骤

1 在 Orchestrator 客户端中单击 **Web Views** 视图。

2 单击 Web 视图标题中的菜单按钮 ( )，然后选择 **New from > File template** 以将 Web 视图模板导入到 Orchestrator 服务器中。

3 浏览到以下文件夹：

`*install_directory*\VMware\Orchestrator\apps\webviewTemplates`

4 选择 `default_webview.zip` 文件，然后单击 **Open**。

5 将 Web 视图命名为 **`Virtual Machine Manager`**。

Virtual Machine Manager Web 视图立即显示在客户端的 **Web Views** 视图的列表中。

6 （可选）右键单击 **`Virtual Machine Manager`**，然后选择 **Publish** 以预览空模板。

7 （可选）在浏览器中，转到 `http://*orchestrator_server*:8280/vmo/`，以查看在 Orchestrator 服务器上运行的 Web 视图的列表。

8 （可选）单击 **Virtual Machine Manager**，并使用 Orchestrator 用户名和密码登录。

您将看到基本的 Web 视图，其中没有任何操作或函数。

您已通过模板创建了一个空的 Web 视图，并可选择在浏览器中检查它。

下一步

将空 Web 视图导出到可以在其中处理它的工作文件夹。

将 Web 视图导出到工作文件夹

要想简化 Web 视图的开发过程，请将工作流模板导出到可在其中处理它的本地文件夹中。

前提条件

必须已将 Web 视图模板导入到 Orchestrator 中，并使用它创建了一个空的 Web 视图。

步骤

1 创建文件夹以在其中开发 Virtual Machine Manager Web 视图。

   例如，`work_dir/`。

2 在 Orchestrator 客户端的 **Web Views** 视图中，右键单击“Virtual Machine Manager”，然后选择 **Export to directory**。

3 导航到工作文件夹，然后单击 **Export**。

   Orchestrator 会在 `work_dir/` 中创建一个包含该 Web 视图的文件夹。例如，如果导出 Virtual Machine Manager Web 视图，Orchestrator 会在工作文件夹中创建名为 `virtual_machine_manager/` 的文件夹。

您已将空 Web 视图导出到工作目录下。

下一步

配置用于 Web 视图开发的 Orchestrator 服务器。

配置 Web 视图开发服务器

可以将 Orchestrator 服务器配置为从工作文件夹发布 Web 视图。此过程使您可以在开发 Web 视图时，不必将它导入 Orchestrator 服务器即可预览该视图。

在 Orchestrator 配置界面中，将 Orchestrator 服务器设置为 Web 视图开发模式。服务器在开发模式下运行时，所有 Web 视图都是从开发文件夹加载而不是从 Orchestrator 服务器的 Web 视图库加载。

注意 因为当服务器处于开发模式时，Orchestrator 会从开发文件夹发布 Web 视图，所以必须将所有 Web 视图导出到同一个开发文件夹中。

前提条件

您必须导入 Web 视图模板，创建 Virtual Machine Manager Web 视图，并将其导出到工作目录中。

步骤

1 在浏览器中打开位于以下 URL 的 Orchestrator 配置界面：

   - `http://`*`orchestrator_server`*`:8282`（如果使用 HTTP 连接到服务器）。
   - `https://`*`orchestrator_server`*`:8283`（如果使用 HTTPS 以安全方式连接到服务器）。

2 使用配置用户名和密码登录配置界面。

   Orchestrator 管理员会在首次安装 Orchestrator 时设置此用户名和密码。

3 在 **General** 视图中单击 **Advanced Configuration** 选项卡。

4 选中 **Webview development enable** 复选框。

5 在文本框中键入指向工作文件夹根目录的路径。

   注意 请确保输入正确的工作文件夹根目录路径。不要在路径中包括包含工作流自身的文件夹。

   ---

6 单击 **Apply Settings** 按钮。

7 单击 **Startup Options** 视图。

8 单击 **Restart Service** 以使用 Web 视图开发模式重新启动 Orchestrator 服务器。

将 Orchestrator 服务器设置为 Web 视图开发模式，此模式允许在开发时预览 Web 视图。

下一步

创建 Web 视图前端页。

导出的 Web 视图内容

要想创建自定义 Web 视图，必须修改 Web 视图模板中包含的部分文件。

导出 Virtual Machine Manager Web 视图之后，*work_dir*/vm_manager 文件夹包含下列文件夹和文件。

/components 包含以下 Tapestry 组件文件。

Border.jwc	用于定义 Web 视图页使用的默认模板组件
Border.html	用于定义 Web 视图页的默认结构

/css 包含以下样式表。

border.css	用于呈现边框模板的样式表
custom.css	可用于自定义其他样式表，比如 Orchestrator 提供的用于呈现表单等复杂组件的 webform.css 或 common.css 样式表。除 Border.html 以外的每个 Web 视图页都要导入 custom.css 文件。custom.css 文件使用您定义的布局覆盖其他样式表。 **注意** custom.css 样式表是唯一可编辑的样式表。如果直接编辑其他系统样式表，而不是编辑 custom.css，每次升级到新版本的 Orchestrator 时，所编辑的内容都将被覆盖。
login.css	可用于自定义登录页。

/images 包含 Web 视图默认外观的所有相关图像。

/default.html Web 视图的主页。

/login.html 用于定义 Web 视图登录页的外观。

/login.page 用于定义 Web 视图登录页的行为。

编辑 Web 视图前端页

在 default.html 文件中创建 Web 视图的前端页。

前提条件

必须已将 Web 视图模板导入到 Orchestrator 中，将其内容导出到工作目录，并在 Web 视图开发模式下配置了 Orchestrator。

步骤

1 转到工作目录的根节点。

2 在 HTML 编辑器中打开 `default.html` 文件。

`default.html` 页使用 `Border` 组件呈现。它包含很少代码，如以下代码示例所示。

```
<!-- Load the border located in ~/components/layout/Border.jwc -->
<vmo jwcid="@layout/MyBorder" section="literal:home" title="Home">
    <!--  Content of the homepage -->
        <h2 style="margin-left:16px; margin-top:0px; padding-top:18px;">
Welcome to Default Webview Template
</h2>
        <p style="margin-left:16px;">
This webview is a base for your own webview development.
</p>
</vmo>
```

`vmo` 标记通过将 `jwcid` 属性设置为指向 `Border` 来初始化 Tapestry 组件。`Border` 组件呈现页面的布局。您放入 `vmo` 标记中的内容呈现在组件的正文中。

下一步

可以将函数添加到 `default.html` 页。

创建用于获取虚拟机的操作

可以将 Virtual Machine Manager Web 视图的前端页自定义为在虚拟文件夹中显示所有虚拟机的列表。必须创建一个操作才能获取虚拟机。

前提条件

必须已将 Web 视图模板导入到 Orchestrator 中，将其内容导出到工作目录，并在 Web 视图开发模式下配置了 Orchestrator。

步骤

1 在 Orchestrator 客户端中单击 **Actions** 视图。

2 右键单击操作层次结构列表的根节点，然后选择 **New module** 以新建操作模块。

3 将新模块命名为 ***org.my_company*.vmManager**。

4 在模块的层次结构列表中右键单击 ***org.my_company*.vmManager**，并选择 **Add Action**。

5 将新操作命名为 **getVmByFolder**。

6 通过右键单击 **getVmByFolder** 并选择 **Edit**，打开操作工作台。

7 单击 **Scripting** 选项卡。

8 右键单击 **Scripting** 选项卡的参数窗格，然后选择 **Add parameter**。

9 将参数命名为 **vmFolder**。

10 单击 **vmFolder** 参数的 **Type** 链接，然后选择 **Array of > VC:VirtualMachine**

11 在 **Scripting** 选项卡的脚本编写面板中添加以下 JavaScript 代码。

```
return vmFolder.vm
```

12 单击 **Return type:Void** 链接设置操作的返回类型。

13 从选取对话框中选择 **Array of: > VC:VirtualMachine**，然后单击 **Accept**。

14 单击 **Save and Close** 关闭操作工作台。

您创建了一个从文件夹返回虚拟机阵列的操作。

下一步

设置 Web 视图授权。

设置 Web 视图授权

通过设置授权，可以定义哪些用户有权访问 Web 视图。

在 Orchestrator 客户端的 **Authorizations** 视图中设置 Web 视图的授权。

前提条件

必须已创建 `getVmByFolder` 操作，以便为 Virtual Machine Manager Web 视图获取一个虚拟机阵列。

步骤

1 在 Orchestrator 客户端中单击 **Authorizations** 视图。

2 单击位于 **Authorizations** 视图标题中的菜单按钮 ()，然后选择 **Create authorization**。

3 在 **Name** 文本框中键入授权的名称。

4 单击 **LDAP Group** 旁的 **Not set**，选择一个 LDAP 用户组以授权其使用该 Web 视图。

5 在 **Description** 文本框中键入授权的描述信息。

6 单击 **Create**。

7 右键单击该授权并选择 **Edit**。

8 单击 **References** 选项卡。

9 单击 **Create reference**。

10 从 vCenter Server 清单层次结构列表中选择一个文件夹，然后单击 **Select** 以授权访问此文件夹。

11 再次单击 **Create reference**，并选择 **getVm** 操作。

12 单击 **Save and Close** 退出授权工作台。

您设置了用于访问 Virtual Machine Manager Web 视图访问的文件夹和操作的权限。

下一步

设置 Web 视图属性。

设置 Web 视图属性

您可以设置Web视图属性以将其指向要执行的操作。通过在Orchestrator客户端中编辑Web视图，可以设置Web视图的属性。

前提条件

必须已创建 `getVmByFolder` 操作，以便为 Virtual Machine Manager Web 视图获取一个虚拟机阵列，并且设置授权以访问它。

步骤

1 在 Orchestrator 客户端中单击 **Web Views** 视图。

2 右键单击 **Virtual Machine Manager**，然后选择 **Unpublish**。

   无法编辑已发布的 Web 视图。

3 右键单击 **Virtual Machine Manager**，然后选择 **Edit**。

4 在 Web 视图工作台上单击 **Attributes** 选项卡。

5 右键单击 **Attributes** 选项卡并选择 **Add attribute**。

6 单击属性名称并将其命名为 **`getVirtualMachineList`**。

   单击 **`getVirtualMachineList`** **Type** 链接，然后选择 **Action**。

7 单击 **`getVirtualMachineList`** **Value** 链接，然后选择之前创建的 **getVmByFolder** 操作。

8 重复步骤 5 到步骤 7 以创建具有以下属性的另一个属性。

   - 名称：**`vmFolder`**
   - 类型：**`VC:VmFolder`**
   - 值：来自清单的包含多个虚拟机的虚拟文件夹。

9 单击 **Save and Close** 退出 Web 视图工作台。

您已定义将 Web 视图指向您所创建的操作及要访问的虚拟机文件夹的属性。

下一步

将组件添加到 Web 视图页以访问您定义的操作。

将组件添加到网页

可以将组件添加到执行耦合操作的 Web 视图页中。

前提条件

必须完成以下任务：

- 创建 `getVmByFolder` 操作为 Virtual Machine Manager Web 视图获取一个虚拟机阵列。
- 设置该视图的访问授权配置。
- 设置用于访问此操作和虚拟机文件夹的 Virtual Machine Manager Web 视图属性。

步骤

1 在 HTML 编辑器中打开 `default.html` Web 视图文件。

2 将 `vso:ListPane Component` 添加到 `default.html` 文件中，以便在页中列出所有虚拟机。

可通过在 `default.html` 文件的 `vmo` 标记之间添加以下代码来添加 `vso:ListPane Component`。

```
<vmo jwcid="@layout/MyBorder" section="literal:home" title="Home">
            <h1>Virtual Machine Manager</h1>
<div jwcid="vmList@vso:ListPane"
action="getVirtualMachineList"
actionParameters="attribute:vmFolder"
detailUrl="<myWebViewComponent>.html">
Select a virtual machine on the left to display it.
</vmo>
```

`vmo` 标记可实例化 `vso:ListPane` 组件，如下所示。

`jwcid="vmList@ListPane"`	`vmList` 是组件的唯一标识符，而 `@ListPane` 则是组件类名称。
`action="getVirtualMachineList"`	链接到包含要执行的操作的名称的 Web 视图属性。
`actionParameters="attribute:vmFolder"`	使用 OGNL 语句将属性传递到操作中。上述代码会将 `vmFolder` Webview 属性传递到操作中。
detailUrl="<*myWebViewComponent*>.html"	所创建的 HTML Web 视图组件模板的路径。

您在 Virtual Machine Manager Web 视图中添加了一个组件，该组件用于执行您定义的从 vCenter Server 的给定文件夹中获取虚拟机的操作。

下一步

自定义界面。

自定义 Web 视图界面

Web 视图开发的最后一个阶段是自定义界面。

前提条件

必须已在 Web 视图前端页中添加了一个组件，用于执行获取虚拟机的操作。

步骤

1 在浏览器中，转到 Virtual Machine Manager Web 视图并刷新页面。

所添加的组件显示在此 Web 视图中，但您必须修改其页面布局。

2 打开 `work_dir/vm_manager/css/custom.css` 样式表文件。

3 将以下代码添加到 work_dir/vm_manager/css/custom.css。

```
.vso_listPane {
border:1px solid #CCC;
height:300px;
margin-bottom:1em;
 }
.vso_listPane .vso_virtualMachine {
padding:9px;
}
```

4 在浏览器中刷新页面。

   该组件具有细边框和固定高度。

通过调整 **custom.css** 样式表，可自定义 Web 视图的外观。

下一步

通过修改 custom.css 文件，可自定义页面的任意部分。覆盖 Border.html 文件和 border.css 可以修改总体布局。您可以使用局部模板和样式表覆盖现有组件的任意外观。可以创建 Tapestry 组件。

升级 vCenter Server 之后重构 Orchestrator 应用程序

9

如果将虚拟基础架构从 VMware Infrastructure 3.5 升级到 vCenter Server 4.0，则必须重构您编写的所有 Orchestrator 应用程序以便与旧版本配合使用。Orchestrator 4.0 提供了可帮助您将应用程序重构为新版本的工作流。另外，您也可以通过安装 VMware Infrastructure 3.5 插件在 Orchestrator 4.0 上运行 VMware Infrastructure 3.5。

本章讨论了以下主题：

何时重构应用程序

如果您将虚拟基础架构从 VMware Infrastructure 3.5 升级到 vCenter Server 4.0，则必须执行相应操作才能继续运行现有的应用程序。

将 Orchestrator 应用程序作为插件开发。插件由一个或多个软件包组成，这些软件包可包含工作流、操作、Java 类、XML 文件、Web 视图、配置元素或策略模板。vCenter Server 4.0 API 使用的软件包名称和特定对象类型与先前版本的 VMware Infrastructure 所使用的不同。名称和类型更改均在表 9-1 中列出。

如果要将虚拟基础架构从 VMware Infrastructure 3.5 升级到 vCenter Server 4.0，其中一种方法是在 Orchestrator Server 4.0 平台上安装 VMware Infrastructure 3.5 插件，然后导入您的应用程序。VMware Infrastructure 3.5 插件可与 vCenter Server 4.0 插件进行通信，使您可以无需更改即可运行这些应用程序。

此外，要想充分利用 vCenter Server 4.0 的所有功能特性，还可以采用另外一种方法，即重构用旧版本编写的 Orchestrator 应用程序。Orchestrator 4.0 提供了可帮助您将应用程序重构为新版本的工作流。

表 9–1 对象名称更改（从先前版本的 VMware Infrastructure 到 vCenter Server 4.0）

对象	先前版本的 VMware Infrastructure 中的值	vCenter Server 4.0 中的值
软件包名称	`vim3`	`vcenter`
`FinderResult` 类型	`VIM3`	`VC`
脚本类型	`Vim`	`Vc`<*脚本名称*>
主机	`VMware3:VimHost`	`VC:SdkConnection`

由于这些名称和类型发生了变化，您必须更新由应用程序执行的所有 VMware Infrastucture 函数调用，以便应用程序可以通过 vCenter Server 4.0 插件在 vCenter Server 4.0 API 软件包中找到这些函数。

重要事项 为了避免在重构期间意外覆盖软件包，必须在运行重构工作流之前备份软件包和应用程序。如果重构失败，请将服务器还原为先前的状态。

此外，重构应用程序时，所有重构的工作流都将获得新的工作流 ID。如果您拥有通过使用其 ID 访问工作流的应用程序（例如，Web 服务客户端），则必须相应地更新这些工作流 ID。

安装 VMware Infrastructure 3.5 插件

要继续运行用 Orchestrator 4.0 为 VMware Infrastructure 3.5 开发的应用程序，一种解决方案是安装 VMware Infrastructure 3.5 可选插件。VMware Infrastructure 3.5 插件随 Orchestrator 4.0 提供，但并不是默认安装的组件。

如果安装了 VMware Infrastructure 3.5 插件，即可运行针对 VMware Infrastructure 3.5 开发的应用程序。但是，使用这些应用程序将无法充分利用 vCenter Server 4.0 的功能特性。要充分利用 vCenter Server 4.0 的功能特性，请重构 VMware Infrastructure 3.5 应用程序。

步骤

1 在浏览器中打开位于以下 URL 的 Orchestrator 配置界面并登录。

   `http://`*<orchestrator_server_ip_地址>*`:8282`

2 在 **General** 选项卡中，单击 **Install Application**。

3 浏览到以下位置之一。

   - `c:\Program Files\VMware\Orchestrator\extras\plugins`（如果安装的是 Orchestrator 独立版本）。
   - `c:\Program Files\VMware\Infrastructure\Orchestrator\extras\plugins`（如果 vCenter Server 安装了 Orchestrator）。

4 选择 `vmo_vi35_4_0_0_4198.vmoapp`，然后单击 **Open**。

5 单击 **Install**。

6 单击 **Startup Options**。

7 单击 **Restart Service**，重新启动 Orchestrator 服务器。

您已安装 VMware Infrastrcuture 3.5 插件。

下一步

可以使用 Orchestrator 4.0 运行为 VMware Infrastrcuture 3.5 编写的应用程序。

使用基本重构工作流重构软件包

Orchestrator 提供基本工作流以帮助您重构大多数软件包，以便这些软件包可以访问 vCenter Server 4.0。

重构工作流会创建一个现有 VMware Infrastructure 3.5 Orchestrator 软件包的副本，然后将副本中的所有元素修改为可使用 vCenter Server 4.0。运行这些工作流时，原始 VMware Infrastructure 3.5 应用程序保持原样不变，但新的副本会更新为可使用 vCenter Server 4.0 运行。

安装 Refactoring Tutorial 示例应用程序（可选）

您可以使用 Refactor Tutorial 示例来体验重构工作流的过程。可以将示例软件包导入到 Orchestrator 中，然后安装该示例。

Refactor Tutorial 示例由一个包含两个基本工作流和一个 Web 视图的软件包组成。

- Suspend virtual machine 工作流请求将 VMware Infrastructure 3.5 清单中运行的某个虚拟机挂起，并等待 VMware Infrastructure 完成挂起请求。
- Resume virtual machine 工作流请求恢复挂起的 VMware Infrastructure 3.5 虚拟机，并等待 VMware Infrastructure 完成恢复请求。
- 通过 Refactor Tutorial Web 视图，您可以从浏览器的 Web 界面中运行 Suspend virtual machine 和 Resume virtual machine 工作流。

最初，Refactor Tutorial Web 视图会访问 VMware Infrastructure 3.5 清单中正在运行的虚拟机，并与其进行交互。运行重构工作流之后，此 Web 视图应用程序会转而访问 vCenter Server 4.0 清单中的虚拟机，并与其进行交互。

前提条件

- 必须已经在 Orchestrator 服务器上安装了 VMware Infrastructure 3.5 和 vCenter Server 4.0 插件。
- Orchestrator 必须已连接到 VMware Infrastructure 3.5 服务器和 vCenter Server 4.0 服务器，并且这两个服务器中都有虚拟机正在运行。
- 必须已经从 Orchestrator 文档索引页面下载示例的 ZIP 文件。请参见第 5 页，“示例应用程序”。

步骤

1 将示例的 ZIP 文件解压缩到适当的位置。

2 打开 `<安装_目录>/Refactoring`。

   `Refactoring` 文件夹包含 `com.vmware.refactor.tutorial.package`。

3 导入 `com.vmware.refactor.tutorial.package`。

   a 在 Orchestrator 客户端的左侧窗格中单击 **Packages** 视图。

   b 右键单击软件包列表下的任意空白位置，然后选择 **Import Package**。

   c 找到 `com.vmware.library.refactor.tutorial.package`，然后单击 **Open**。

      此时会显示软件包证书的相关信息。

   d 单击 **Import**。

      即会显示软件包内容，并包含导入元素的对照表。

   e 单击 **Import checked elements**。

4 选择 **Documentation > Refactor tutorial**，检查 **Workflows** 视图的左侧窗格中是否显示 Refactoring Tutorial 工作流。

5 启动 Refactor Tutorial Web 视图。

   a 在左侧窗格中，单击 **Web Views** 视图。

   b 右键单击 Refactor Tutorial Web 视图，然后选择 **Publish** 以启动该视图。

6 在 Web 浏览器中，请转至 `http://`*`orchestrator_server`*`:8280`，其中 *orchestrator_server* 是 Orchestrator 服务器正在其上运行的计算机的名称或 IP 地址。

7 选择 **Web View List > Refactor Tutorial**。

8 使用您访问 Orchestrator 客户端时所用的用户名和密码登录。

您已经安装了 Refactor Tutorial 示例。通过 Refactor Tutorial Web 视图，您可以挂起和恢复 VMware Infrastructure 3.5 服务器中正在运行的虚拟机。您可以体验将应用程序从 VMware Infrastucture 3.5 重构为 vCenter Server 4.0 的过程。

下一步

可以重构此示例应用程序。

运行基本重构工作流

基本重构工作流可成功重构大多数 VMware Infrastructure 3.5 应用程序，以便这些应用程序可以使用 vCenter Server 4.0 API。

前提条件

- 必须具有一个要重构的 VMware Infrastructure 3.5 Orchestrator 应用程序。例如，`com.vmware.refactor.tutorial` 示例程序。

步骤

1 在 Orchestrator 客户端界面的左侧窗格中单击 **Workflows** 视图。

2 在工作流层次结构列表中选择 **Library > Refactoring**，查看重构工作流。

3 右键单击 Migrate package to vCenter Server 4 工作流，然后选择 **Execute Workflow**。

4 在软件包下方，单击 **Source package** 值，并选择此软件包以将其从 VMware Infrastructure 3.5 重构为 vCenter Server 4.0 软件包。

   例如，`com.vmware.refactor.tutorial`。

5 在 **Destination package** 文本框中输入目标软件包。

   使用目标软件包创建重构的应用程序。

   例如，`com.vmware.refactor.tutorial_vcenter40`。

6 在 **Rules** 下方，将重构工作流指向应用程序包含的其他对象集，并提供要在其中复制这些对象的目标。

   如果不提供目标，则重构工具不会执行重构。如果应用程序不具有某个特定类型的对象，请将该文本框的设置保留为 **Not set**。例如，重构教程示例中包含一个 Web 视图对象和一个工作流对象，因此需要设置以下源位置和目标位置。

规则类型	源位置	目标位置
Configuration rules	**Not set**	无
Webview rules	**Refactor Tutorial**	`refactortutorial_copy`
Workflow rules	**Refactor tutorial**	`Documentation/Refactor tutorial Copy`
Actions rules	**Not set**	无

   重构教程示例不包含配置元素或操作，因此无须设置这些值。但是，对于包含配置元素或操作的应用程序，必须提供源位置和目标位置。

   注意 设置操作规则的源位置时，不能从列表选择位置。必须手动键入源位置。

7 在“Save XML”下方，单击 **Not set** 链接，选择要在其中存储重构工作流结果的资源类别。

8 单击 **Submit**。

Migrate package to vCenter Server 4 工作流即会运行。

Migrate package to vCenter Server 4 工作流已复制并已重构应用程序，以便新版本实施 vCenter Server 4.0 插件，而不是 VMware Infrastructure 3.5 插件。VMware Infrastructure 3.5 版应用程序保留在原有位置，未做修改。

下一步

验证重构工作流已正确重构应用程序。

验证重构

运行重构工作流后，请验证工作流是否正确重构了应用程序。

步骤

1 在 Orchestrator 客户端界面中单击 **Packages** 视图，检查新的软件包是否存在。

在 Refactoring Tutorial 示例中，列出了名为 `com.vmware.refactor.tutorial_vcenter40` 的软件包。

2 单击 **Workflows** 视图，检查重构的工作流是否存在。

在 Refactoring Tutorial 示例中，选择 **Documentation > Refactor tutorial copy**，查看工作流层次结构列表中的 Submit VM 和 Resume VM 工作流。

3 检查新工作流实施 vCenter Server 4.0 插件的情况。

a 单击其中一个新工作流。

b 在右侧窗格中，单击 **Schema** 选项卡。

c 在架构图中，双击其中一个元素以在左侧的元素层次结构列表中显示该元素。

vCenter Server 4.0 插件会显示在此列表中。

例如，在 Refactor Tutorial 示例中，选择 **Refactor Tutorial Copy > Suspend VM**，然后双击 **Suspend virtual machine and wait** 工作流元素。Suspend virtual machine and wait 工作流元素位于工作流层次结构列表的 **Library > vCenter Server > Virtual machine management > Power Management** 节点上。在示例的原始版本中，此元素位于 VIM3 节点上，而不是 vCenter Server 节点。

4 单击 **Web views** 视图。

5 右键单击旧版 Refactor Tutorial Web 视图，然后选择 **Unpublish**。

6 右键单击已重构的新 Refactor Tutorial Web 视图，然后选择 **Publish**。

7 在浏览器中，请转至 `http://`*<orchestrator_server>*`:8280/vmo/refactortutorial/default.html`，访问 Refactor Tutorial Web 视图。

8 使用您的 Orchestrator 用户名和密码登录 Refactor Tutorial Web 视图。

9 单击 **Suspend VM**。

10 搜索要挂起的虚拟机。

如果可以挂起的虚拟机处于 vCenter Server 4.0 环境中，则重构成功。

经验证，您已成功重构应用程序。现在，您的应用程序可实施 vCenter Server 4.0 插件。

使用高级重构工作流重构软件包

如果基本重构工作流无法成功重构 VMware Infrastructure 3.5 应用程序，则可以尝试使用高级重构工作流来重构此应用程序。

高级重构工作流使用的工作流与基本重构工作流调用的工作流相同。如果基本工作流无效，或者您要自己定义重构工作流的输出参数，可以直接使用高级重构工作流。

高级重构工作流会创建一个现有 VMware Infrastructure 3.5 Orchestrator 应用程序的副本，然后将副本中的所有元素修改为可使用 vCenter Server 4.0。运行这些工作流时，原始 VMware Infrastructure 3.5 应用程序保持原样不变，但新的副本会更新为可使用 vCenter Server 4.0 运行。

高级重构工作流

高级重构工作流使用 XML 描述文件来定义要重构的应用程序、应用程序副本的创建位置、应用程序中要重构的元素以及重构规则的查找位置等。高级重构工作流中包括一个为您创建这些 XML 描述文件的工作流。

表 9–2 显示 Orchestrator 提供的高级重构工作流。

表 9–2 高级重构工作流

工作流名称	描述
Library > Refactoring > Refactor - Create RefactorDescription XML	请求用户输入以定义下列信息： ■ 要重构的应用程序的名称和位置 ■ 软件包对象的复制规则 ■ 要在应用程序副本中重构的元素类型 ■ 重构工作流使用的资源文件的名称和位置 此类信息记录在 XML 描述文件中，其他重构工作流在运行时会使用这些文件。
Library > Refactoring > Advanced > Refactor - Copy VMware Infrastructure 3.5 application and migrate to vCenter Server 4.0 plug-in	执行以下操作： ■ 使用 XML 描述文件中指定的名称和位置创建应用程序的副本。 ■ 更新应用程序副本中的元素，以便这些元素实施 vCenter Server 4.0 插件。 ■ 提供已通过工作流更新的所有元素的日志（可选）。此日志是 XML 映射文件，您可以在重构其他应用程序时使用此文件。
Library > Refactoring > Advanced > Refactor - Copy VMware Infrastructure 3.5 application and migrate to vCenter Server 4.0 plug-in with String input	执行以下操作： ■ 使用您在 XML 描述文件中以字符串形式提供给工作流的指定名称和位置创建应用程序的副本。 ■ 更新应用程序副本中的元素，以便这些元素实施 vCenter Server 4.0 插件。 ■ 提供已通过工作流更新的所有元素的日志（可选）。此日志是 XML 映射文件，您可以在重构其他应用程序时使用此文件。
Library > Refactoring > Advanced > Refactor - Execute refactor	在没有输入参数的情况下执行重构。此工作流由 Refactor - Copy VMware Infrastructure 3.5 application and migrate to vCenter Server 4.0 plug-in 工作流运行。如有必要，可以修改此工作流以定义特定于应用程序的重构规则。这是高级用例。
Library > Refactoring > Advanced > Refactor - Execute refactor using resources	根据资源元素（即 XML 文件）中定义的规则（而不是 XML 字符串中定义的规则）执行重构。此工作流由 Refactor - Copy VMware Infrastructure 3.5 application and migrate to vCenter Server 4.0 plug-in 工作流运行。如有必要，可以修改此工作流以定义特定于应用程序的重构规则。这是高级用例。

运行高级重构工作流

如果基本的 Migrate Package to vCenter 4.0 工作流无法成功重构应用程序，您可以运行高级重构工作流。

前提条件

- 必须已经在 Orchestrator 服务器上安装了 VMware Infrastructure 3.5 和 vCenter Server 4.0 插件。
- 必须已经在 Orchestrator 客户端中导入 `com.vmware.library.refactoring` 软件包。
- 必须具有一个要重构的 VMware Infrastructure 3.5 Orchestrator 应用程序。例如，`com.vmware.refactor.tutorial` 示例程序。

步骤

1. 运行 Create RefactorDescription XML 工作流，创建一个 `copy.xml` XML 描述文件。
2. 再次运行 Create RefactorDescription XML 工作流，创建一个 `update-references.xml` XML 描述文件。
3. 运行 Copy VMware Infrastructure 3.5 application to refactor the application and migrate to the vCenter Server 4.0 plug-in 工作流。

   要运行重构工作流，必须向其传递 `copy.xml` 文件和 `update-references.xml` 文件。

您已通过直接运行高级重构工作流重构应用程序。

索引

V

W

X

Y